Oracle® Fusion Middleware

Developing JAX-RPC Web Services for Oracle WebLogic Server

12*c* (12.1.2)

E28140-01

June 2013

Documentation for software developers that describes how to develop WebLogic Web services using Java API for XML-based RPC (JAX-RPC).

ORACLE

Oracle Fusion Middleware Developing JAX-RPC Web Services for Oracle WebLogic Server, 12*c* (12.1.2)

E28140-01

Contents

4 Programming the JWS File

5 Understanding Data Binding

6 Developing JAX-RPC Web Service Clients

7 Invoking a Web Service Using Asynchronous Request-Response

8 Using Web Services Reliable Messaging

9 Creating Conversational Web Services

10 Creating Buffered Web Services

11 Using the Asynchronous Features Together

12 Using Callbacks to Notify Clients of Events

13 Using JMS Transport as the Connection Protocol

14 Creating and Using SOAP Message Handlers

Preface

This preface describes the document accessibility features and conventions used in this guide—*Developing JAX-RPC Web Services for Oracle WebLogic Server*.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What's New in This Guide

The following topics introduce the new and changed features of WebLogic Java API for XML-based RPC (JAX-RPC) Web services in Oracle Fusion Middleware Release 12*c* (12.1.2), and provides pointers to additional information.

For Release 12*c* (12.1.2), the contents of the following guides, delivered in Oracle Fusion Middleware Release 11*g*, have been merged into this single document:

- *Getting Started With JAX-RPC Web Services for Oracle WebLogic Server*
- *Programming Advanced Features of JAX-RPC Web Services for Oracle WebLogic Server*

New and Changed Features for Release 12*c* (12.1.2)

Oracle Fusion Middleware Release 12*c* (12.1.2) includes the following new and changed features for this document:

- New standalone Web service client JAR file that supports basic JAX-RPC client-side functionality. See Section 6.4, "Using a Standalone Client JAR File When Invoking Web Services".

Part I

Introduction

Part I introduces developing WebLogic Web services using the Java API for XML-based RPC (JAX-RPC).

Sections include:

1

Introduction to JAX-RPC Web Services

This chapter provides a summary table of topics for software developers who program WebLogic Web services using Java API for XML-based RPC (JAX-RPC).

JAX-RPC is a specification that defines the Java APIs for making XML-based remote procedure calls (RPC). In particular, these APIs are used to invoke and get a response from a Web service using SOAP 1.1, and XML-based protocol for exchange of information in a decentralized and distributed environment. For more information, see `http://java.net/projects/jax-rpc/`.

The following table summarizes the contents of this guide.

Table 1–1 Content Summary

This section . . .	Describes how to . . .
Chapter 2, "Examples for JAX-RPC Web Service Developers"	Review and run common use cases and examples.
Chapter 3, "Developing JAX-RPC Web Services"	Develop Web services using the WebLogic development environment.
Chapter 4, "Programming the JWS File"	Program the JWS file that implements your Web service.
Chapter 5, "Understanding Data Binding"	Use the Java Architecture for XML Binding (JAXB) data binding.
Chapter 6, "Developing JAX-RPC Web Service Clients"	Invoke your Web service from a Java client or another Web service.
Chapter 7, "Invoking a Web Service Using Asynchronous Request-Response"	Invoke a Web service asynchronously.
Chapter 8, "Using Web Services Reliable Messaging"	Create a reliable Web service, as specified by the WS-ReliableMessaging specification, and then create a client Web services that invokes the reliable Web service.
Chapter 9, "Creating Conversational Web Services"	Create a conversational Web service which communicates with a client.
Chapter 10, "Creating Buffered Web Services"	Create a buffered Web service, which is a simpler type of reliable Web service that one specified by the WS-ReliableMessaging specification.
Chapter 11, "Using the Asynchronous Features Together"	Use the asynchronous features, such as reliable messaging, asynchronous request-response, and conversations, together in a single Web service.
Chapter 12, "Using Callbacks to Notify Clients of Events"	Notify a client of a Web service that an event has happened by programming a callback.

Table 1–1 (Cont.) Content Summary

This section . . .	Describes how to . . .
Chapter 13, "Using JMS Transport as the Connection Protocol"	Specify that JMS, rather than the default HTTP/S, is the connection protocol when invoking a Web service.
Chapter 14, "Creating and Using SOAP Message Handlers"	Create and configure SOAP message handlers for a Web service.
Chapter 15, "Using Database Web Services"	Create a database Web service.

For an overview of WebLogic Web services, standards, samples, and related documentation, see *Understanding WebLogic Web Services for Oracle WebLogic Server*.

For information about WebLogic Web service security, see *Securing WebLogic Web Services for Oracle WebLogic Server*.

2

Examples for JAX-RPC Web Service Developers

This chapter describes common use cases and examples for WebLogic Web services using Java API for XML-based RPC (JAX-RPC).

This chapter includes the following topics:

- Section 2.1, "Creating a Simple HelloWorld Web Service"
- Section 2.2, "Creating a Web Service With User-Defined Data Types"
- Section 2.3, "Creating a Web Service from a WSDL File"
- Section 2.4, "Invoking a Web Service from a Java SE Client"
- Section 2.5, "Invoking a Web Service from a WebLogic Web Service"

Each use case provides step-by-step procedures for creating simple WebLogic Web services and invoking an operation from a deployed Web service. The examples include basic Java code and Ant `build.xml` files that you can use in your own development environment to recreate the example, or by following the instructions to create and run the examples in an environment that is separate from your development environment.

The use cases do not go into detail about the processes and tools used in the examples; later chapters are referenced for more detail.

2.1 Creating a Simple HelloWorld Web Service

This section describes how to create a very simple Web service that contains a single operation. The *Java Web Service (JWS)* file that implements the Web service uses just the one required *JWS annotation*: `@WebService`. A JWS file is a standard Java file that uses JWS metadata annotations to specify the shape of the Web service. Metadata annotations were introduced with JDK 5.0, and the set of annotations used to annotate Web service files are called JWS annotations. WebLogic Web services use standard JWS annotations. For a complete list of JWS annotations that are supported, see "Web Service Annotation Support" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

The following example shows how to create a Web service called `HelloWorldService` that includes a single operation, `sayHelloWorld`. For simplicity, the operation returns the inputted String value.

1. Set your WebLogic Server environment.

   Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the `bin` subdirectory of your

domain directory. The default location of WebLogic Server domains is *ORACLE_HOME*/user_projects/domains/*domainName*, where *ORACLE_HOME* is the directory you specified as Oracle Home when you installed Oracle WebLogic Server and *domainName* is the name of your domain.

2. Create a project directory, as follows:

```
prompt> mkdir /myExamples/hello_world
```

3. Create a `src` directory under the project directory, as well as subdirectories that correspond to the package name of the JWS file (shown later in this procedure):

```
prompt> cd /myExamples/hello_world
 prompt> mkdir src/examples/webservices/hello_world
```

4. Create the JWS file that implements the Web service.

   Open your favorite Java IDE or text editor and create a Java file called `HelloWorldImpl.java` using the Java code specified in Section 2.1.1, "Sample HelloWorldImpl.java JWS File."

   The sample JWS file shows a Java class called `HelloWorldImpl` that contains a single public method, `sayHelloWorld(String)`. The `@WebService` annotation specifies that the Java class implements a Web service called `HelloWorldService`. By default, all public methods are exposed as operations.

5. Save the `HelloWorldImpl.java` file in the `src/examples/webservices/hello_world` directory.

6. Create a standard Ant `build.xml` file in the project directory (`myExamples/hello_world/src`) and add a `taskdef` Ant task to specify the full Java classname of the `jwsc` task:

```
<project name="webservices-hello_world" default="all">
  <taskdef name="jwsc"
           classname="weblogic.wsee.tools.anttasks.JwscTask" />
</project>
```

   See Section 2.1.2, "Sample Ant Build File for HelloWorldImpl.java" for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`, `undeploy`, `client`, and `run`. The full `build.xml` file also uses properties, such as `${ear-dir}`, rather than always using the hard-coded name for the EAR directory.

7. Add the following call to the `jwsc` Ant task to the `build.xml` file, wrapped inside of the `build-service` target:

```
<target name="build-service">
  <jwsc
     srcdir="src"
     destdir="output/helloWorldEar">
       <jws file="examples/webservices/hello_world/HelloWorldImpl.java"
          type="JAXRPC"/>
  </jwsc>
</target>
```

   The `jwsc` WebLogic Web service Ant task generates the supporting artifacts (such as the deployment descriptors, serialization classes for any user-defined data types, the WSDL file, and so on), compiles the user-created and generated Java code, and archives all the artifacts into an Enterprise Application EAR file that you later deploy to WebLogic Server.

8. Execute the `jwsc` Ant task by specifying the `build-service` target at the command line:

   ```
   prompt> ant build-service
   ```

   See the `output/helloWorldEar` directory to view the files and artifacts generated by the `jwsc` Ant task.

9. Start the WebLogic Server instance to which the Web service will be deployed.

10. Deploy the Web service, packaged in an enterprise application, to WebLogic Server, using either the Administration Console or the `wldeploy` Ant task. In either case, you deploy the `helloWorldEar` Enterprise application, located in the `output` directory.

    To use the `wldeploy` Ant task, add the following target to the `build.xml` file:

    ```
    <taskdef name="wldeploy"
             classname="weblogic.ant.taskdefs.management.WLDeploy"/>
    <target name="deploy">
      <wldeploy action="deploy"
                name="helloWorldEar" source="output/helloWorldEar"
                user="${wls.username}" password="${wls.password}"
                verbose="true"
                adminurl="t3://${wls.hostname}:${wls.port}"
                targets="${wls.server.name}" />
    </target>
    ```

    Substitute the values for `wls.username`, `wls.password`, `wls.hostname`, `wls.port`, and `wls.server.name` that correspond to your WebLogic Server instance.

    Deploy the WAR file by executing the `deploy` target:

    ```
    prompt> ant deploy
    ```

11. Test that the Web service is deployed correctly by invoking its WSDL in your browser:

    ```
    http://host:port/HelloWorldImpl/HelloWorldImpl?WSDL
    ```

    You construct the URL using the values of the `contextPath` and `serviceUri` attributes of the `WLHttpTransport` JWS annotation; however, because the JWS file in this use case does not include the `WLHttpTransport` annotation, use the default values for the `contextPath` and `serviceUri` attributes: the name of the Java class in the JWS file. These attributes will be set explicitly in the next example, Section 2.2, "Creating a Web Service With User-Defined Data Types." Use the hostname and port relevant to your WebLogic Server instance.

You can use the `clean`, `build-service`, `undeploy`, and `deploy` targets in the `build.xml` file to iteratively update, rebuild, undeploy, and redeploy the Web service as part of your development process.

To run the Web service, you need to create a client that invokes it. See Section 2.4, "Invoking a Web Service from a Java SE Client" for an example of creating a Java client application that invokes a Web service.

2.1.1 Sample HelloWorldImpl.java JWS File

```
package examples.webservices.hello_world;
// Import the @WebService annotation
import javax.jws.WebService;
```

```
@WebService(name="HelloWorldPortType", serviceName="HelloWorldService")
/**
  * This JWS file forms the basis of simple Java-class implemented WebLogic
  * Web Service with a single operation: sayHelloWorld
  */
public class HelloWorldImpl {
   // By default, all public methods are exposed as Web Services operation
   public String sayHelloWorld(String message) {
   try {
     System.out.println("sayHelloWorld:" + message);
   } catch (Exception ex) { ex.printStackTrace(); }

     return "Here is the message: '" + message + "'";
   }
 }
```

2.1.2 Sample Ant Build File for HelloWorldImpl.java

The following `build.xml` file uses properties to simplify the file.

```
<project name="webservices-hello_world" default="all">
  <!-- set global properties for this build -->
  <property name="wls.username" value="weblogic" />
   <property name="wls.password" value="weblogic" />
   <property name="wls.hostname" value="localhost" />
   <property name="wls.port" value="7001" />
   <property name="wls.server.name" value="myserver" />
  <property name="ear.deployed.name" value="helloWorldEar" />
   <property name="example-output" value="output" />
   <property name="ear-dir" value="${example-output}/helloWorldEar" />
   <property name="clientclass-dir" value="${example-output}/clientclasses" />
  <path id="client.class.path">
     <pathelement path="${clientclass-dir}"/>
     <pathelement path="${java.class.path}"/>
   </path>
  <taskdef name="jwsc"
     classname="weblogic.wsee.tools.anttasks.JwscTask" />
  <taskdef name="clientgen"
     classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
  <taskdef name="wldeploy"
     classname="weblogic.ant.taskdefs.management.WLDeploy"/>
   <target name="all" depends="clean,build-service,deploy,client" />
  <target name="clean" depends="undeploy">
     <delete dir="${example-output}"/>
   </target>
  <target name="build-service">
    <jwsc
       srcdir="src"
       destdir="${ear-dir}">
      <jws file="examples/webservices/hello_world/HelloWorldImpl.java"
           type="JAXRPC"/>
    </jwsc>
  </target>
  <target name="deploy">
     <wldeploy action="deploy" name="${ear.deployed.name}"
       source="${ear-dir}" user="${wls.username}"
       password="${wls.password}" verbose="true"
       adminurl="t3://${wls.hostname}:${wls.port}"
       targets="${wls.server.name}" />
```

```
   </target>
  <target name="undeploy">
     <wldeploy action="undeploy" name="${ear.deployed.name}"
       failonerror="false"
       user="${wls.username}" password="${wls.password}" verbose="true"
       adminurl="t3://${wls.hostname}:${wls.port}"
       targets="${wls.server.name}" />
   </target>
  <target name="client">
    <clientgen

wsdl="http://${wls.hostname}:${wls.port}/HelloWorldImpl/HelloWorldImpl?WSDL"
       destDir="${clientclass-dir}"
       packageName="examples.webservices.hello_world.client"
       type="JAXRPC"/>
    <javac
       srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
       includes="**/*.java"/>
    <javac
       srcdir="src" destdir="${clientclass-dir}"
       includes="examples/webservices/hello_world/client/**/*.java"/>
  </target>
  <target name="run">
     <java classname="examples.webservices.hello_world.client.Main"
           fork="true" failonerror="true" >
       <classpath refid="client.class.path"/>
       <arg
  line="http://${wls.hostname}:${wls.port}/HelloWorldImpl/HelloWorldImpl"  />
     </java> </target>
</project>
```

2.2 Creating a Web Service With User-Defined Data Types

The preceding use case uses only a simple data type, `String`, as the parameter and return value of the Web service operation. This next example shows how to create a Web service that uses a user-defined data type, in particular a JavaBean called `BasicStruct`, as both a parameter and a return value of its operation.

There is actually very little a programmer has to do to use a user-defined data type in a Web service, other than to create the Java source of the data type and use it correctly in the JWS file. The `jwsc` Ant task, when it encounters a user-defined data type in the JWS file, automatically generates all the data binding artifacts needed to convert data between its XML representation (used in the SOAP messages) and its Java representation (used in WebLogic Server). The data binding artifacts include the XML Schema equivalent of the Java user-defined type, the JAX-RPC type mapping file, and so on.

The following procedure is very similar to the procedure in Section 2.1, "Creating a Simple HelloWorld Web Service." For this reason, although the procedure does show all the needed steps, it provides details only for those steps that differ from the simple HelloWorld example.

1. Set your WebLogic Server environment.

   Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`ORACLE_HOME`*`/user_projects/domains/`*`domainName`*, where *`ORACLE_HOME`* is the

directory you specified as Oracle Home when you installed Oracle WebLogic Server and *domainName* is the name of your domain.

2. Create a project directory:

```
prompt> mkdir /myExamples/complex
```

3. Create a `src` directory under the project directory, as well as subdirectories that correspond to the package name of the JWS file (shown later in this procedure):

```
prompt> cd /myExamples/complex
prompt> mkdir src/examples/webservices/complex
```

4. Create the source for the `BasicStruct` JavaBean.

   Open your favorite Java IDE or text editor and create a Java file called `BasicStruct.java`, in the project directory, using the Java code specified in Section 2.2.1, "Sample BasicStruct JavaBean."

5. Save the `BasicStruct.java` file in the `src/examples/webservices/complex` subdirectory of the project directory.

6. Create the JWS file that implements the Web service using the Java code specified in Section 2.2.2, "Sample ComplexImpl.java JWS File."

   The sample JWS file uses several JWS annotations: `@WebMethod` to specify explicitly that a method should be exposed as a Web service operation and to change its operation name from the default method name `echoStruct` to `echoComplexType`; `@WebParam` and `@WebResult` to configure the parameters and return values; `@SOAPBinding` to specify the type of Web service; and `@WLHttpTransport` to specify the URI used to invoke the Web service. The `ComplexImpl.java` JWS file also imports the `examples.webservice.complex.BasicStruct` class and then uses the `BasicStruct` user-defined data type as both a parameter and return value of the `echoStruct()` method.

   For more in-depth information about creating a JWS file, see Chapter 4, "Programming the JWS File."

7. Save the `ComplexImpl.java` file in the `src/examples/webservices/complex` subdirectory of the project directory.

8. Create a standard Ant `build.xml` file in the project directory and add a `taskdef` Ant task to specify the fully Java classname of the `jwsc` task:

```
<project name="webservices-complex" default="all">
  <taskdef name="jwsc"
           classname="weblogic.wsee.tools.anttasks.JwscTask" />
</project>
```

   See Section 2.2.3, "Sample Ant Build File for ComplexImpl.java JWS File" for a full sample `build.xml` file.

9. Add the following call to the `jwsc` Ant task to the `build.xml` file, wrapped inside of the `build-service` target:

```
<target name="build-service">
  <jwsc
     srcdir="src"
     destdir="output/ComplexServiceEar" >
     <jws file="examples/webservices/complex/ComplexImpl.java"
        type="JAXRPC">
       <WLHttpTransport
```

```
        contextPath="complex" serviceUri="ComplexService"
        portName="ComplexServicePort"/>
      </jws>
    </jwsc>
</target>
```

In the preceding example:

- The `type` attribute of the `<jws>` element specifies the type of Web service (JAX-WS or JAX-RPC).
- The `<WLHttpTransport>` child element of the `<jws>` element of the `jwsc` Ant task specifies the context path and service URI sections of the URL used to invoke the Web service over the HTTP/S transport, as well as the name of the port in the generated WSDL. This value overrides the value specified in the JWS file using the `@WLHttpTransport` attribute. For more information about defining the context path, see "Defining the Context Path of a WebLogic Web Service" on page 3-9.

10. Execute the `jwsc` Ant task:

    ```
    prompt> ant build-service
    ```

    See the `output/ComplexServiceEar` directory to view the files and artifacts generated by the `jwsc` Ant task.

11. Start the WebLogic Server instance to which the Web service will be deployed.

12. Deploy the Web service, packaged in the `ComplexServiceEar` Enterprise Application, to WebLogic Server, using either the Administration Console or the `wldeploy` Ant task. For example:

    ```
    prompt> ant deploy
    ```

13. Deploy the Web service, packaged in an Enterprise Application, to WebLogic Server, using either the Administration Console or the `wldeploy` Ant task. In either case, you deploy the `ComplexServiceEar` Enterprise application, located in the `output` directory.

    To use the `wldeploy` Ant task, add the following target to the `build.xml` file:

    ```
    <taskdef name="wldeploy"
             classname="weblogic.ant.taskdefs.management.WLDeploy"/>
    <target name="deploy">
      <wldeploy action="deploy"
                name="ComplexServiceEar" source="output/ComplexServiceEar"
                user="${wls.username}" password="${wls.password}"
                verbose="true"
                adminurl="t3://${wls.hostname}:${wls.port}"
                targets="${wls.server.name}" />
    </target>
    ```

    Substitute the values for `wls.username`, `wls.password`, `wls.hostname`, `wls.port`, and `wls.server.name` that correspond to your WebLogic Server instance.

    Deploy the WAR file by executing the `deploy` target:

    ```
    prompt> ant deploy
    ```

14. Test that the Web service is deployed correctly by invoking its WSDL in your browser:

```
http://host:port/complex/ComplexService?WSDL
```

To run the Web service, you need to create a client that invokes it. See Section 2.4, "Invoking a Web Service from a Java SE Client" for an example of creating a Java client application that invokes a Web service.

2.2.1 Sample BasicStruct JavaBean

```
package examples.webservices.complex;
/**
  * Defines a simple JavaBean called BasicStruct that has integer, String,
  * and String[] properties
  */
public class BasicStruct {
  // Properties
  private int intValue;
   private String stringValue;
   private String[] stringArray;
  // Getter and setter methods
  public int getIntValue() {
     return intValue;
   }
  public void setIntValue(int intValue) {
     this.intValue = intValue;
   }
  public String getStringValue() {
     return stringValue;
   }
  public void setStringValue(String stringValue) {
     this.stringValue = stringValue;
   }
  public String[] getStringArray() {
     return stringArray;
   }
  public void setStringArray(String[] stringArray) {
     this.stringArray = stringArray;
   }
  public String toString() {
     return "IntValue="+intValue+", StringValue="+stringValue;
   }
 }
```

2.2.2 Sample ComplexImpl.java JWS File

```
package examples.webservices.complex;
// Import the standard JWS annotation interfaces
import javax.jws.WebMethod;
 import javax.jws.WebParam;
 import javax.jws.WebResult;
 import javax.jws.WebService;
 import javax.jws.soap.SOAPBinding;
// Import the WebLogic-specific JWS annotation interface
import weblogic.jws.WLHttpTransport;
// Import the BasicStruct JavaBean
import examples.webservices.complex.BasicStruct;
// Standard JWS annotation that specifies that the portType name of the Web
 // Service is "ComplexPortType", its public service name is "ComplexService",
 // and the targetNamespace used in the generated WSDL is "http://example.org"
@WebService(serviceName="ComplexService", name="ComplexPortType",
```

```
            targetNamespace="http://example.org")
// Standard JWS annotation that specifies this is a document-literal-wrapped
 // Web Service
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
             use=SOAPBinding.Use.LITERAL,
             parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)
// WebLogic-specific JWS annotation that specifies the context path and service
 // URI used to build the URI of the Web Service is "complex/ComplexService"
@WLHttpTransport(contextPath="complex", serviceUri="ComplexService",
                 portName="ComplexServicePort")
/**
  * This JWS file forms the basis of a WebLogic Web Service.  The Web Services
  * has two public operations:
  *
  *  - echoInt(int)
  *  - echoComplexType(BasicStruct)
  *
  * The Web Service is defined as a "document-literal" service, which means
  * that the SOAP messages have a single part referencing an XML Schema element
  * that defines the entire body.
  */
public class ComplexImpl {
  // Standard JWS annotation that specifies that the method should be exposed
   // as a public operation.  Because the annotation does not include the
   // member-value "operationName", the public name of the operation is the
   // same as the method name: echoInt.
   //
   // The WebResult annotation specifies that the name of the result of the
   // operation in the generated WSDL is "IntegerOutput", rather than the
   // default name "return".   The WebParam annotation specifies that the input
   // parameter name in the WSDL file is "IntegerInput" rather than the Java
   // name of the parameter, "input".
  @WebMethod()
   @WebResult(name="IntegerOutput",
              targetNamespace="http://example.org/complex")
   public int echoInt(
       @WebParam(name="IntegerInput",
                 targetNamespace="http://example.org/complex")
       int input)
  {
     System.out.println("echoInt '" + input + "' to you too!");
     return input;
   }
  // Standard JWS annotation to expose method "echoStruct" as a public operation
   // called "echoComplexType"
   // The WebResult annotation specifies that the name of the result of the
   // operation in the generated WSDL is "EchoStructReturnMessage",
   // rather than the default name "return".
  @WebMethod(operationName="echoComplexType")
   @WebResult(name="EchoStructReturnMessage",
              targetNamespace="http://example.org/complex")
   public BasicStruct echoStruct(BasicStruct struct)
  {
     System.out.println("echoComplexType called");
     return struct;
   }
 }
```

2.2.3 Sample Ant Build File for ComplexImpl.java JWS File

The following `build.xml` file uses properties to simplify the file.

```
<project name="webservices-complex" default="all">
  <!-- set global properties for this build -->
  <property name="wls.username" value="weblogic" />
   <property name="wls.password" value="weblogic" />
   <property name="wls.hostname" value="localhost" />
   <property name="wls.port" value="7001" />
   <property name="wls.server.name" value="myserver" />
  <property name="ear.deployed.name" value="complexServiceEAR" />
   <property name="example-output" value="output" />
   <property name="ear-dir" value="${example-output}/complexServiceEar" />
   <property name="clientclass-dir" value="${example-output}/clientclass" />
  <path id="client.class.path">
     <pathelement path="${clientclass-dir}"/>
     <pathelement path="${java.class.path}"/>
   </path>
  <taskdef name="jwsc"
     classname="weblogic.wsee.tools.anttasks.JwscTask" />
  <taskdef name="clientgen"
     classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
  <taskdef name="wldeploy"
     classname="weblogic.ant.taskdefs.management.WLDeploy"/>
  <target name="all" depends="clean,build-service,deploy,client"/>
  <target name="clean" depends="undeploy">
     <delete dir="${example-output}"/>
   </target>
  <target name="build-service">
    <jwsc
       srcdir="src"
       destdir="${ear-dir}"
       keepGenerated="true"
       >
       <jws file="examples/webservices/complex/ComplexImpl.java"
          type="JAXRPC">
         <WLHttpTransport
           contextPath="complex" serviceUri="ComplexService"
           portName="ComplexServicePort"/>
       </jws>
    </jwsc>
  </target>
  <target name="deploy">
     <wldeploy action="deploy"
       name="${ear.deployed.name}"
       source="${ear-dir}" user="${wls.username}"
       password="${wls.password}" verbose="true"
       adminurl="t3://${wls.hostname}:${wls.port}"
       targets="${wls.server.name}"/>
   </target>
  <target name="undeploy">
     <wldeploy action="undeploy" failonerror="false"
       name="${ear.deployed.name}"
       user="${wls.username}" password="${wls.password}" verbose="true"
       adminurl="t3://${wls.hostname}:${wls.port}"
       targets="${wls.server.name}"/>
   </target>
  <target name="client">
    <clientgen
       wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
```

```
      destDir="${clientclass-dir}"
      packageName="examples.webservices.complex.client"
         type="JAXRPC"/>
    <javac
      srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
      includes="**/*.java"/>
    <javac
      srcdir="src" destdir="${clientclass-dir}"
      includes="examples/webservices/complex/client/**/*.java"/>
   </target>
  <target name="run" >
     <java fork="true"
          classname="examples.webservices.complex.client.Main"
          failonerror="true" >
      <classpath refid="client.class.path"/>
      <arg line="http://${wls.hostname}:${wls.port}/complex/ComplexService"
  />
     </java>
   </target>
</project>
```

2.3 Creating a Web Service from a WSDL File

Another common use case of creating a Web service is to start from an existing WSDL file, often referred to as the *golden WSDL*. A WSDL file is a public contract that specifies what the Web service looks like, such as the list of supported operations, the signature and shape of each operation, the protocols and transports that can be used when invoking the operations, and the XML Schema data types that are used when transporting the data. Based on this WSDL file, you generate the artifacts that implement the Web service so that it can be deployed to WebLogic Server. You use the `wsdlc` Ant task to generate the following artifacts.

- JWS service endpoint interface (SEI) that implements the Web service described by the WSDL file.
- JWS implementation file that contains a partial (stubbed-out) implementation of the generated JWS SEI. This file must be customized by the developer.
- Data binding artifacts used by WebLogic Server to convert between the XML and Java representations of the Web service parameters and return values.
- Optional Javadocs for the generated JWS SEI.

> **Note:** The only file generated by the `wsdlc` Ant task that you update is the JWS implementation file. You never need to update the JAR file that contains the JWS SEI and data binding artifacts.

Typically, you run the `wsdlc` Ant task one time to generate a JAR file that contains the generated JWS SEI file and data binding artifacts, then code the generated JWS file that implements the interface, adding the business logic of your Web service. In particular, you add Java code to the methods that implement the Web service operations so that the operations behave as needed and add additional JWS annotations.

After you have coded the JWS implementation file, you run the `jwsc` Ant task to generate the deployable Web service, using the same steps as described in the preceding sections. The only difference is that you use the `compiledWsdl` attribute to

specify the JAR file (containing the JWS SEI file and data binding artifacts) generated by the `wsdlc` Ant task.

The following simple example shows how to create a Web service from the WSDL file shown in Section 2.3.1, "Sample WSDL File." The Web service has one operation, `getTemp`, that returns a temperature when passed a zip code.

1. Set your WebLogic Server environment.

   Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`ORACLE_HOME`*`/user_projects/domains/`*`domainName`*, where *`ORACLE_HOME`* is the directory you specified as Oracle Home when you installed Oracle WebLogic Server and *`domainName`* is the name of your domain.

2. Create a working directory:

   ```
   prompt> mkdir /myExamples/wsdlc
   ```

3. Put your WSDL file into an accessible directory on your computer.

   For the purposes of this example, it is assumed that your WSDL file is called `TemperatureService.wsdl` and is located in the `/myExamples/wsdlc/wsdl_files` directory. See Section 2.3.1, "Sample WSDL File" for a full listing of the file.

4. Create a standard Ant `build.xml` file in the project directory and add a `taskdef` Ant task to specify the full Java classname of the `wsdlc` task:

   ```
   <project name="webservices-wsdlc" default="all">
     <taskdef name="wsdlc"
              classname="weblogic.wsee.tools.anttasks.WsdlcTask"/>
   </project>
   ```

   See Section 2.3.3, "Sample Ant Build File for TemperatureService" for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`, `undeploy`, `client`, and `run`. The full `build.xml` file also uses properties, such as `${ear-dir}`, rather than always using the hard-coded name for the EAR directory.

5. Add the following call to the `wsdlc` Ant task to the `build.xml` file, wrapped inside of the `generate-from-wsdl` target:

   ```
   <target name="generate-from-wsdl">
     <wsdlc
          srcWsdl="wsdl_files/TemperatureService.wsdl"
          destJwsDir="output/compiledWsdl"
          destImplDir="output/impl"
          packageName="examples.webservices.wsdlc" />
   </target>
   ```

   The `wsdlc` task in the examples generates the JAR file that contains the JWS SEI and data binding artifacts into the `output/compiledWsdl` directory under the current directory. It also generates a partial implementation file (`TemperaturePortTypeImpl.java`) of the JWS SEI into the `output/impl/examples/webservices/wsdlc` directory (which is a combination of the output directory specified by `destImplDir` and the directory hierarchy specified by the package name). All generated JWS files will be packaged in the `examples.webservices.wsdlc` package.

6. Execute the `wsdlc` Ant task by specifying the `generate-from-wsdl` target at the command line:

```
prompt> ant generate-from-wsdl
```

   See the `output` directory if you want to examine the artifacts and files generated by the `wsdlc` Ant task.

7. Update the generated `output/impl/examples/webservices/wsdlc/TemperaturePortTypeImpl.java` JWS implementation file using your favorite Java IDE or text editor to add Java code to the methods so that they behave as you want.

   See Section 2.3.2, "Sample TemperaturePortType Java Implementation File" for an example; the added Java code is in **bold**. The generated JWS implementation file automatically includes values for the `@WebService` and `@WLHttpTransport` JWS annotations that correspond to the values in the original WSDL file.

   Note: There are restrictions on the JWS annotations that you can add to the JWS implementation file in the "starting from WSDL" use case. See "wsdlc" in the *WebLogic Web Services Reference for Oracle WebLogic Server* for details.

   For simplicity, the sample `getTemp()` method in `TemperaturePortTypeImpl.java` returns a hard-coded number. In real life, the implementation of this method would actually look up the current temperature at the given zip code.

8. Copy the updated `TemperaturePortTypeImpl.java` file into a permanent directory, such as a `src` directory under the project directory; remember to create child directories that correspond to the package name:

```
prompt> cd /examples/wsdlc
 prompt> mkdir src/examples/webservices/wsdlc
 prompt> cp output/impl/examples/webservices/wsdlc/TemperaturePortTypeImpl.java
\
            src/examples/webservices/wsdlc/TemperaturePortTypeImpl.java
```

9. Add a `build-service` target to the `build.xml` file that executes the `jwsc` Ant task against the updated JWS implementation class. Use the `compiledWsdl` attribute of `jwsc` to specify the name of the JAR file generated by the `wsdlc` Ant task:

```
<taskdef name="jwsc"
   classname="weblogic.wsee.tools.anttasks.JwscTask" />
<target name="build-service">
  <jwsc
     srcdir="src"
     destdir="${ear-dir}">
    <jws file="examples/webservices/wsdlc/TemperaturePortTypeImpl.java"
          compiledWsdl="${compiledWsdl-dir}/TemperatureService_wsdl.jar"
          type="JAXRPC">
        <WLHttpTransport
         contextPath="temp" serviceUri="TemperatureService"
         portName="TemperaturePort">
     </WLHttpTransport>
    </jws>
  </jwsc>
</target>
```

In the preceding example:

- The `type` attribute of the `<jws>` element specifies the type of Web services (JAX-WS or JAX-RPC).
- The `<WLHttpTransport>` child element of the `<jws>` element of the `jwsc` Ant task specifies the context path and service URI sections of the URL used to invoke the Web service over the HTTP/S transport, as well as the name of the port in the generated WSDL. This value overrides the value specified in the JWS file using the `@WLHttpTransport` attribute.

10. Execute the `build-service` target to generate a deployable Web service:

```
prompt> ant build-service
```

You can re-run this target if you want to update and then re-build the JWS file.

11. Start the WebLogic Server instance to which the Web service will be deployed.
12. Deploy the Web service, packaged in an Enterprise Application, to WebLogic Server, using either the Administration Console or the `wldeploy` Ant task. In either case, you deploy the `wsdlcEar` Enterprise application, located in the `output` directory.

    To use the `wldeploy` Ant task, add the following target to the `build.xml` file:

```
<taskdef name="wldeploy"
          classname="weblogic.ant.taskdefs.management.WLDeploy"/>
<target name="deploy">
  <wldeploy action="deploy" name="wsdlcEar"
     source="output/wsdlcEar" user="${wls.username}"
     password="${wls.password}" verbose="true"
     adminurl="t3://${wls.hostname}:${wls.port}"
     targets="${wls.server.name}" />
</target>
```

    Substitute the values for `wls.username`, `wls.password`, `wls.hostname`, `wls.port`, and `wls.server.name` that correspond to your WebLogic Server instance.

    Deploy the WAR file by executing the `deploy` target:

```
prompt> ant deploy
```

13. Test that the Web service is deployed correctly by invoking its WSDL in your browser:

```
http://host:port/temp/TemperatureService?WSDL
```

    The context path and service URI section of the preceding URL are specified by the original golden WSDL. Use the hostname and port relevant to your WebLogic Server instance. Note that the deployed and original WSDL files are the same, except for the host and port of the endpoint address.

You can use the `clean`, `build-service`, `undeploy`, and `deploy` targets in the `build.xml` file to iteratively update, rebuild, undeploy, and redeploy the Web service as part of your development process.

To run the Web service, you need to create a client that invokes it. See Section 2.4, "Invoking a Web Service from a Java SE Client" for an example of creating a Java client application that invokes a Web service.

2.3.1 Sample WSDL File

```
<?xml version="1.0"?>
<definitions
    name="TemperatureService"
    targetNamespace="http://www.xmethods.net/sd/TemperatureService.wsdl"
    xmlns:tns="http://www.xmethods.net/sd/TemperatureService.wsdl"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/" >
        <message name="getTempRequest">
                <part name="zip" type="xsd:string"/>
         </message>
        <message name="getTempResponse">
                <part name="return" type="xsd:float"/>
         </message>
        <portType name="TemperaturePortType">
                <operation name="getTemp">
                        <input message="tns:getTempRequest"/>
                        <output message="tns:getTempResponse"/>
                </operation>
         </portType>
        <binding name="TemperatureBinding" type="tns:TemperaturePortType">
                <soap:binding style="document"
                             transport="http://schemas.xmlsoap.org/soap/http"/>
                <operation name="getTemp">
                        <soap:operation soapAction=""/>
                        <input>
                                <soap:body use="literal"
                                namespace="urn:xmethods-Temperature" />
                        </input>
                        <output>
                                <soap:body use="literal"
                                namespace="urn:xmethods-Temperature" />
                        </output>
                </operation>
         </binding>
        <service name="TemperatureService">
             <documentation>
                 Returns current temperature in a given U.S. zipcode
             </documentation>
             <port name="TemperaturePort" binding="tns:TemperatureBinding">
                <soap:address

location="http://localhost:7001/temp/TemperatureService"/>
             </port>
         </service>
</definitions>
```

2.3.2 Sample TemperaturePortType Java Implementation File

```
package examples.webservices.wsdlc;
import javax.jws.WebService;
 import weblogic.jws.*;
/**
  * TemperaturePortTypeImpl class implements web service endpoint
  * interface TemperaturePortType */
@WebService(
   serviceName="TemperatureService",
```

```
   targetNamespace="http://www.xmethods.net/sd/TemperatureService.wsdl"
   endpointInterface="examples.webservices.wsdlc.TemperaturePortType)
@WLHttpTransport(
   contextPath="temp",
   serviceUri="TemperatureService",
   portName="TemperaturePort")
 public class TemperaturePortTypeImpl implements
examples.webservices.wsdlc.TemperaturePortType {
  public TemperaturePortTypeImpl() { }
  public float getTemp(java.lang.String zip) {
      return 1.234f;
   }
 }
```

2.3.3 Sample Ant Build File for TemperatureService

The following `build.xml` file uses properties to simplify the file.

```
<project default="all">
  <!-- set global properties for this build -->
  <property name="wls.username" value="weblogic" />
   <property name="wls.password" value="weblogic" />
   <property name="wls.hostname" value="localhost" />
   <property name="wls.port" value="7001" />
   <property name="wls.server.name" value="myserver" />
  <property name="ear.deployed.name" value="wsdlcEar" />
   <property name="example-output" value="output" />
   <property name="compiledWsdl-dir" value="${example-output}/compiledWsdl" />
   <property name="impl-dir" value="${example-output}/impl" />
   <property name="ear-dir" value="${example-output}/wsdlcEar" />
   <property name="clientclass-dir" value="${example-output}/clientclasses" />
  <path id="client.class.path">
     <pathelement path="${clientclass-dir}"/>
     <pathelement path="${java.class.path}"/>
   </path>
  <taskdef name="wsdlc"
            classname="weblogic.wsee.tools.anttasks.WsdlcTask"/>
  <taskdef name="jwsc"
     classname="weblogic.wsee.tools.anttasks.JwscTask" />
  <taskdef name="clientgen"
     classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
  <taskdef name="wldeploy"
     classname="weblogic.ant.taskdefs.management.WLDeploy"/>
  <target name="all"
     depends="clean,generate-from-wsdl,build-service,deploy,client" />
  <target name="clean" depends="undeploy">
     <delete dir="${example-output}"/>
   </target>
  <target name="generate-from-wsdl">
    <wsdlc
         srcWsdl="wsdl_files/TemperatureService.wsdl"
         destJwsDir="${compiledWsdl-dir}"
         destImplDir="${impl-dir}"
         packageName="examples.webservices.wsdlc" />
  </target>
  <target name="build-service">
    <jwsc
       srcdir="src"
       destdir="${ear-dir}">
```

```
      <jws file="examples/webservices/wsdlc/TemperaturePortTypeImpl.java"
            compiledWsdl="${compiledWsdl-dir}/TemperatureService_wsdl.jar"
            type="JAXRPC">
          <WLHttpTransport
           contextPath="temp" serviceUri="TemperatureService"
           portName="TemperaturePort"/>
       </jws>
    </jwsc>
  </target>
  <target name="deploy">
     <wldeploy action="deploy" name="${ear.deployed.name}"
       source="${ear-dir}" user="${wls.username}"
       password="${wls.password}" verbose="true"
       adminurl="t3://${wls.hostname}:${wls.port}"
       targets="${wls.server.name}" />
   </target>
  <target name="undeploy">
     <wldeploy action="undeploy" name="${ear.deployed.name}"
       failonerror="false"
       user="${wls.username}" password="${wls.password}" verbose="true"
       adminurl="t3://${wls.hostname}:${wls.port}"
       targets="${wls.server.name}" />
   </target>
  <target name="client">
    <clientgen
       wsdl="http://${wls.hostname}:${wls.port}/temp/TemperatureService?WSDL"
       destDir="${clientclass-dir}"
       packageName="examples.webservices.wsdlc.client"
       type="JAXRPC">
     <javac
       srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
       includes="**/*.java"/>
    <javac
       srcdir="src" destdir="${clientclass-dir}"
       includes="examples/webservices/wsdlc/client/**/*.java"/>
  </target>
  <target name="run">
     <java classname="examples.webservices.wsdlc.client.TemperatureClient"
          fork="true" failonerror="true" >
       <classpath refid="client.class.path"/>
       <arg
           line="http://${wls.hostname}:${wls.port}/temp/TemperatureService" />
     </java>
  </target>
</project>
```

2.4 Invoking a Web Service from a Java SE Client

Note: As described in this section, you can invoke a Web service from any Java SE or Java EE application running on WebLogic Server (with access to the WebLogic Server classpath). For information about support for *stand-alone* Java applications that are running in an environment where WebLogic Server libraries are not available, see Section 6.4, "Using a Standalone Client JAR File When Invoking Web Services".

When you invoke an operation of a deployed Web service from a client application, the Web service could be deployed to WebLogic Server or to any other application server, such as .NET. All you need to know is the URL to its public contract file, or WSDL.

In addition to writing the Java client application, you must also run the `clientgen` WebLogic Web service Ant task to generate the artifacts that your client application needs to invoke the Web service operation. These artifacts include:

- The Java class for the JAX-RPC `Stub` and `Service` interface implementations for the particular Web service you want to invoke.
- The Java class for any user-defined XML Schema data types included in the WSDL file.
- The JAX-RPC mapping deployment descriptor file which contains information about the mapping between the Java user-defined data types and their corresponding XML Schema types in the WSDL file.
- A client-side copy of the WSDL file.

The following example shows how to create a Java client application that invokes the `echoComplexType` operation of the `ComplexService` WebLogic Web service described in Section 2.2, "Creating a Web Service With User-Defined Data Types." The `echoComplexType` operation takes as both a parameter and return type the `BasicStruct` user-defined data type.

> **Note:** It is assumed in this procedure that you have created and deployed the `ComplexService` Web service.

1. Set your WebLogic Server environment.

   Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`ORACLE_HOME`*`/user_projects/domains/`*`domainName`*, where *`ORACLE_HOME`* is the directory you specified as Oracle Home when you installed Oracle WebLogic Server and *`domainName`* is the name of your domain.

2. Create a project directory:

   ```
   prompt> mkdir /myExamples/simple_client
   ```

3. Create a `src` directory under the project directory, as well as subdirectories that correspond to the package name of the Java client application (shown later on in this procedure):

   ```
   prompt> cd /myExamples/simple_client
    prompt> mkdir src/examples/webservices/simple_client
   ```

4. Create a standard Ant `build.xml` file in the project directory and add a `taskdef` Ant task to specify the full Java classname of the `clientgen` task:

   ```
   <project name="webservices-simple_client" default="all">
     <taskdef name="clientgen"
        classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
   </project>
   ```

   See Section 2.4.2, "Sample Ant Build File For Building Java Client Application" for a full sample `build.xml` file. The full `build.xml` file uses properties, such as

`${clientclass-dir}`, rather than always using the hard-coded name output directory for client classes.

5. Add the following calls to the `clientgen` and `javac` Ant tasks to the `build.xml` file, wrapped inside of the `build-client` target:

```
  <target name="build-client">
    <clientgen
       wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
       destDir="output/clientclass"
       packageName="examples.webservices.simple_client"
       type="JAXRPC"/>
    <javac
       srcdir="output/clientclass" destdir="output/clientclass"
       includes="**/*.java"/>
 <javac
     srcdir="src" destdir="output/clientclass"
     includes="examples/webservices/simple_client/*.java"/>
</target>
```

   The `clientgen` Ant task uses the WSDL of the deployed `ComplexService` Web service to generate the necessary artifacts and puts them into the `output/clientclass` directory, using the specified package name. Replace the variables with the actual hostname and port of your WebLogic Server instance that is hosting the Web service.

   The `clientgen` Ant task also automatically generates the `examples.webservices.complex.BasicStruct` JavaBean class, which is the Java representation of the user-defined data type specified in the WSDL.

   The `build-client` target also specifies the standard `javac` Ant task, in addition to `clientgen`, to compile all the Java code, including the simple Java program described in the next step, into class files.

   The `clientgen` Ant task also provides the `destFile` attribute if you want the Ant task to automatically compile the generated Java code and package all artifacts into a JAR file. For details and an example, see "clientgen" in the *WebLogic Web Services Reference for Oracle WebLogic Server*.

6. Create the Java client application file that invokes the `echoComplexType` operation.

   Open your favorite Java IDE or text editor and create a Java file called `Main.java` using the code specified in Section 2.4.1, "Sample Java Client Application."

   The `Main` client application takes a single argument: the WSDL URL of the Web service. The application then follows standard JAX-RPC guidelines to invoke an operation of the Web service using the Web service-specific implementation of the `Service` interface generated by `clientgen`. The application also imports and uses the `BasicStruct` user-defined type, generated by the `clientgen` Ant task, that is used as a parameter and return value for the `echoStruct` operation. For details, see Chapter 6, "Developing JAX-RPC Web Service Clients."

7. Save the `Main.java` file in the `src/examples/webservices/simple_client` subdirectory of the main project directory.

8. Execute the `clientgen` and `javac` Ant tasks by specifying the `build-client` target at the command line:

```
prompt> ant build-client
```

See the `output/clientclass` directory to view the files and artifacts generated by the `clientgen` Ant task.

9. Add the following targets to the `build.xml` file, used to execute the `Main` application:

```
<path id="client.class.path">
   <pathelement path="output/clientclass"/>
   <pathelement path="${java.class.path}"/>
 </path>
<target name="run" >
  <java fork="true"
        classname="examples.webservices.simple_client.Main"
        failonerror="true" >
     <classpath refid="client.class.path"/>
     <arg line="http://${wls.hostname}:${wls.port}/complex/ComplexService" />
   </java>
</target>
```

The `run` target invokes the `Main` application, passing it the WSDL URL of the deployed Web service as its single argument. The `classpath` element adds the `clientclass` directory to the CLASSPATH, using the reference created with the `<path>` task.

10. Execute the `run` target to invoke the `echoComplexType` operation:

```
prompt> ant run
```

If the invoke was successful, you should see the following final output:

```
run:
     [java] echoComplexType called. Result: 999, Hello Struct
```

You can use the `build-client` and `run` targets in the `build.xml` file to iteratively update, rebuild, and run the Java client application as part of your development process.

2.4.1 Sample Java Client Application

The following provides a simple Java client application that invokes the `echoComplexType` operation.

```
package examples.webservices.simple_client;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
// import the BasicStruct class, used as a param and return value of the
 // echoComplexType operation. The class is generated automatically by
 // the clientgen Ant task.
import examples.webservices.complex.BasicStruct;
/**
  * This is a simple Java client application that invokes the
  * echoComplexType operation of the ComplexService Web service.
  */
public class Main {
  public static void main(String[] args)
       throws ServiceException, RemoteException {
    ComplexService service = new ComplexService_Impl (args[0] + "?WSDL" );
     ComplexPortType port = service.getComplexServicePort();
    BasicStruct in = new BasicStruct();
    in.setIntValue(999);
     in.setStringValue("Hello Struct");
```

```
    BasicStruct result = port.echoComplexType(in);
     System.out.println("echoComplexType called. Result: " + result.getIntValue()
+ ", " + result.getStringValue());
   }
 }
```

2.4.2 Sample Ant Build File For Building Java Client Application

The following `build.xml` file defines tasks to build the Java client application. The example uses properties to simplify the file.

```
<project name="webservices-simple_client" default="all">
  <!-- set global properties for this build -->
  <property name="wls.hostname" value="localhost" />
   <property name="wls.port" value="7001" />
  <property name="example-output" value="output" />
   <property name="clientclass-dir" value="${example-output}/clientclass" />
  <path id="client.class.path">
     <pathelement path="${clientclass-dir}"/>
     <pathelement path="${java.class.path}"/>
   </path>
  <taskdef name="clientgen"
     classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
  <target name="clean" >
     <delete dir="${clientclass-dir}"/>
   </target>
  <target name="all" depends="clean,build-client,run" />
  <target name="build-client">
    <clientgen
      type="JAXRPC"
      wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
       destDir="${clientclass-dir}"
       packageName="examples.webservices.simple_client"/>
    <javac
       srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
       includes="**/*.java"/>
    <javac
       srcdir="src" destdir="${clientclass-dir}"
       includes="examples/webservices/simple_client/*.java"/>
   </target>
  <target name="run" >
     <java fork="true"
           classname="examples.webservices.simple_client.Main"
           failonerror="true" >
       <classpath refid="client.class.path"/>
    <arg line="http://${wls.hostname}:${wls.port}/complex/ComplexService" />
     </java>
   </target>
</project>
```

2.5 Invoking a Web Service from a WebLogic Web Service

You can also invoke a Web service (WebLogic, .NET, and so on) from within a deployed WebLogic Web service.

The procedure for invoking a Web service from a WebLogic Web service is similar to that described in Section 2.4, "Invoking a Web Service from a Java SE Client" except that instead of running the `clientgen` Ant task to generate the client stubs, you use

the <clientgen> child element of <jws>, inside of the jwsc Ant task. The jwsc Ant task automatically packages the generated client stubs in the invoking Web service WAR file so that the Web service has immediate access to them. You then follow standard JAX-RPC programming guidelines in the JWS file that implements the Web service that invokes the other Web service.

The following example shows how to write a JWS file that invokes the echoComplexType operation of the ComplexService Web service described in Section 2.2, "Creating a Web Service With User-Defined Data Types."

Note: It is assumed that you have successfully deployed the ComplexService Web service.

---

1. Set your WebLogic Server environment.

   Open a command window and execute the setDomainEnv.cmd (Windows) or setDomainEnv.sh (UNIX) script, located in the bin subdirectory of your domain directory. The default location of WebLogic Server domains is *ORACLE_HOME*/user_projects/domains/*domainName*, where *ORACLE_HOME* is the directory you specified as Oracle Home when you installed Oracle WebLogic Server and *domainName* is the name of your domain.

2. Create a project directory:

   ```
   prompt> mkdir /myExamples/service_to_service
   ```

3. Create a src directory under the project directory, as well as subdirectories that correspond to the package name of the JWS and client application files (shown later on in this procedure):

   ```
   prompt> cd /myExamples/service_to_service
    prompt> mkdir src/examples/webservices/service_to_service
   ```

4. Create the JWS file that implements the Web service that invokes the ComplexService Web service.

   Open your favorite Java IDE or text editor and create a Java file called ClientServiceImpl.java using the Java code specified in Section 2.5.1, "Sample ClientServiceImpl.java JWS File."

   The sample JWS file shows a Java class called ClientServiceImpl that contains a single public method, callComplexService(). The Java class imports the JAX-RPC stubs, generated later on by the jwsc Ant task, as well as the BasicStruct Java Bean (also generated by clientgen), which is the data type of the parameter and return value of the echoComplexType operation of the ComplexService Web service.

   The ClientServiceImpl Java class defines one method, callComplexService(), which takes two parameters: a BasicStruct which is passed on to the echoComplexType operation of the ComplexService Web service, and the URL of the ComplexService Web service. The method then uses the standard JAX-RPC APIs to get the Service and PortType of the ComplexService, using the stubs generated by jwsc, and then invokes the echoComplexType operation.

5. Save the ClientServiceImpl.java file in the src/examples/webservices/service_to_service directory.

6. Create a standard Ant `build.xml` file in the project directory and add the following task:

```
<project name="webservices-service_to_service" default="all">
  <taskdef name="jwsc"
     classname="weblogic.wsee.tools.anttasks.JwscTask" />
</project>
```

   The `taskdef` task defines the full classname of the `jwsc` Ant task.

   See Section 2.5.2, "Sample Ant Build File For Building ClientService" for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`, `deploy`, `undeploy`, `client`, and `run`. The full `build.xml` file also uses properties, such as `${ear-dir}`, rather than always using the hard-coded name for the EAR directory.

7. Add the following call to the `jwsc` Ant task to the `build.xml` file, wrapped inside of the `build-service` target:

```
<target name="build-service">
  <jwsc
     srcdir="src"
     destdir="output/ClientServiceEar" >
     <jws
       file="examples/webservices/service_to_service/ClientServiceImpl.java"
      type="JAXRPC">
            <WLHttpTransport
              contextPath="ClientService" serviceUri="ClientService"
              portName="ClientServicePort"/>
     <clientgen
        type="JAXRPC"
 wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
        packageName="examples.webservices.complex" />
     </jws>
   </jwsc>
 </target>
```

   In the preceding example, the `<clientgen>` child element of the `<jws>` element of the `jwsc` Ant task specifies that, in addition to compiling the JWS file, `jwsc` should also generate and compile the client artifacts needed to invoke the Web service described by the WSDL file.

   In this example, the package name is set to `examples.webservices.complex`, which is different from the client application package name, `examples.webservices.simple_client`. As a result, you need to import the appropriate class files in the client application:

```
import examples.webservices.complex.BasicStruct;
 import examples.webservices.complex.ComplexPortType;
 import examples.webservices.complex.ComplexService;
```

   If the package name is set to the same package name as the client application, the import calls would be optional.

8. Execute the `jwsc` Ant task by specifying the `build-service` target at the command line:

```
prompt> ant build-service
```

9. Start the WebLogic Server instance to which you will deploy the Web service.

10. Deploy the Web service, packaged in an enterprise application, to WebLogic Server, using either the Administration Console or the `wldeploy` Ant task. In either case, you deploy the `ClientServiceEar` Enterprise application, located in the `output` directory.

    To use the `wldeploy` Ant task, add the following target to the `build.xml` file:

    ```
    <taskdef name="wldeploy"
             classname="weblogic.ant.taskdefs.management.WLDeploy"/>
    <target name="deploy">
      <wldeploy action="deploy" name="ClientServiceEar"
         source="ClientServiceEar" user="${wls.username}"
         password="${wls.password}" verbose="true"
         adminurl="t3://${wls.hostname}:${wls.port}"
         targets="${wls.server.name}" />
    </target>
    ```

    Substitute the values for `wls.username`, `wls.password`, `wls.hostname`, `wls.port`, and `wls.server.name` that correspond to your WebLogic Server instance.

    Deploy the WAR file by executing the `deploy` target:

    ```
    prompt> ant deploy
    ```

11. Test that the Web service is deployed correctly by invoking its WSDL in your browser:

    ```
    http://host:port/ClientService/ClientService?WSDL
    ```

See Section 2.4, "Invoking a Web Service from a Java SE Client" for an example of creating a Java client application that invokes a Web service.

2.5.1 Sample ClientServiceImpl.java JWS File

The following provides a simple Web service client application that invokes the `echoComplexType` operation.

```
package examples.webservices.service_to_service;
import java.rmi.RemoteException;
 import javax.xml.rpc.ServiceException;
import javax.jws.WebService;
 import javax.jws.WebMethod;
import weblogic.jws.WLHttpTransport;
// Import the BasicStruct data type, generated by clientgen and used
 // by the ComplexService Web Service
import examples.webservices.complex.BasicStruct;
// Import the JAX-RPC Stubs for invoking the ComplexService Web Service.
 // Stubs generated by clientgen
import examples.webservices.service_to_service.ComplexPortType;
 import examples.webservices.service_to_service.ComplexService_Impl;
 import examples.webservices.service_to_service.ComplexService;
@WebService(name="ClientPortType", serviceName="ClientService",
             targetNamespace="http://examples.org")
@WLHttpTransport(contextPath="ClientService", serviceUri="ClientService",
                  portName="ClientServicePort")
public class ClientServiceImpl {
  @WebMethod()
   public String callComplexService(BasicStruct input, String serviceUrl)
       throws ServiceException, RemoteException
   {
```

```
    // Create service and port stubs to invoke ComplexService
     ComplexService service = new ComplexService_Impl(serviceUrl + "?WSDL");
     ComplexPortType port = service.getComplexServicePort();
    // Invoke the echoComplexType operation of ComplexService
     BasicStruct result = port.echoComplexType(input);
     System.out.println("Invoked ComplexPortType.echoComplexType." );
    return "Invoke went okay!  Here's the result: '" + result.getIntValue() + ",
" + result.getStringValue() + "'";
  }
 }
```

2.5.2 Sample Ant Build File For Building ClientService

The following `build.xml` file defines tasks to build the client application. The example uses properties to simplify the file.

The following `build.xml` file uses properties to simplify the file.

```
<project name="webservices-service_to_service" default="all">
  <!-- set global properties for this build -->
  <property name="wls.username" value="weblogic" />
   <property name="wls.password" value="weblogic" />
   <property name="wls.hostname" value="localhost" />
   <property name="wls.port" value="7001" />
   <property name="wls.server.name" value="myserver" />
  <property name="ear.deployed.name" value="ClientServiceEar" />
   <property name="example-output" value="output" />
   <property name="ear-dir" value="${example-output}/ClientServiceEar" />
   <property name="clientclass-dir" value="${example-output}/clientclasses" />
  <path id="client.class.path">
     <pathelement path="${clientclass-dir}"/>
     <pathelement path="${java.class.path}"/>
   </path>
  <taskdef name="jwsc"
     classname="weblogic.wsee.tools.anttasks.JwscTask" />
  <taskdef name="clientgen"
     classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
  <taskdef name="wldeploy"
     classname="weblogic.ant.taskdefs.management.WLDeploy"/>
  <target name="all" depends="clean,build-service,deploy,client" />
  <target name="clean" depends="undeploy">
     <delete dir="${example-output}"/>
   </target>
  <target name="build-service">
    <jwsc
         srcdir="src"
         destdir="${ear-dir}" >
       <jws
          file="examples/webservices/service_to_service/ClientServiceImpl.java"
          type="JAXRPC">
         <WLHttpTransport
           contextPath="ClientService" serviceUri="ClientService"
           portName="ClientServicePort"/>
         <clientgen
 type="JAXRPC"
            wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
                 packageName="examples.webservices.complex" />
       </jws>
    </jwsc>
  </target>
```

```
  <target name="deploy">
     <wldeploy action="deploy" name="${ear.deployed.name}"
       source="${ear-dir}" user="${wls.username}"
       password="${wls.password}" verbose="true"
       adminurl="t3://${wls.hostname}:${wls.port}"
       targets="${wls.server.name}" />
   </target>
  <target name="undeploy">
     <wldeploy action="undeploy" name="${ear.deployed.name}"
       failonerror="false"
       user="${wls.username}"
       password="${wls.password}" verbose="true"
       adminurl="t3://${wls.hostname}:${wls.port}"
       targets="${wls.server.name}" />
   </target>
  <target name="client">
    <clientgen
       wsdl="http://${wls.hostname}:${wls.port}/ClientService/ClientService?WSDL"
       destDir="${clientclass-dir}"
       packageName="examples.webservices.service_to_service.client"
       type="JAXRPC"/>
    <javac
       srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
       includes="**/*.java"/>
    <javac
       srcdir="src" destdir="${clientclass-dir}"
       includes="examples/webservices/service_to_service/client/**/*.java"/>
  </target>
  <target name="run">
     <java classname="examples.webservices.service_to_service.client.Main"
           fork="true"
           failonerror="true" >
           <classpath refid="client.class.path"/>
           <arg

line="http://${wls.hostname}:${wls.port}/ClientService/ClientService"/>
    </java>
  </target>
</project>
```

Part II

Developing Basic JAX-RPC Web Services

Part II describes how to develop basic WebLogic Web services using Java API for XML-based RPC (JAX-RPC).

Sections include:

3 Developing JAX-RPC Web Services

This chapter describes the iterative development process for WebLogic Web services using Java API for XML-based RPC (JAX-RPC).

This chapter includes the following topics:

3.1 Overview of the WebLogic Web Service Programming Model

The WebLogic Web Services programming model centers around *JWS files*—Java files that use *JWS annotations* to specify the shape and behavior of the Web Service—and Ant tasks that execute on the JWS file. JWS annotations are based on the metadata feature, introduced in Version 5.0 of the JDK (specified by JSR-175 at `http://www.jcp.org/en/jsr/detail?id=175`) and include standard annotations defined by *Web Services Metadata for the Java Platform* specification (JSR-181), described at `http://www.jcp.org/en/jsr/detail?id=181`, as well as additional ones. For a complete list of JWS annotations that are supported, see "Web Service Annotation Support" in *WebLogic Web Services Reference for Oracle WebLogic Server*. For additional detailed information about this programming model, see *Understanding WebLogic Web Services for Oracle WebLogic Server*.

The following sections describe the high-level steps for iteratively developing a Web Service, either starting from Java or starting from an existing WSDL file:

- Section 3.3, "Developing WebLogic Web Services Starting From Java: Main Steps"
- Section 3.4, "Developing WebLogic Web Services Starting From a WSDL File: Main Steps"

Iterative development refers to setting up your development environment in such a way so that you can repeatedly code, compile, package, deploy, and test a Web Service until it works as you want. The WebLogic Web Service programming model uses Ant tasks to perform most of the steps of the iterative development process. Typically, you create a single `build.xml` file that contains targets for all the steps, then repeatedly run the targets, after you have updated your JWS file with new Java code, to test that the updates work as you expect.

In addition to the command-line tools described in this section, you can use an IDE, such as Oracle JDeveloper, to develop Web services. For more information, see "Using Oracle IDEs to Build Web Services" in *Understanding WebLogic Web Services for Oracle WebLogic Server*.

3.2 Configuring Your Domain For Web Services Features

After you have created a WebLogic Server domain, you can use the Configuration Wizard to update the domain, using a Web Services-specific extension template, so that the resources required by certain WebLogic Web Services features are automatically configured. Although use of this extension template is not required, it makes the configuration of JMS and JDBC resources much easier.

The Web Services extension template automatically configures the resources required for the following features:

- Web Services Reliable Messaging
- Buffering
- JMS Transport

> **Note:** A domain that does not contain Web Services resources will still boot and operate correctly for non-Web services scenarios, and any Web Services scenario that does not involve asynchronous request and response. You will, however, see INFO messages in the server log indicating that asynchronous resources have not been configured and that the asynchronous response service for Web services has not been completely deployed.

The following procedures describe how to create and extend a domain so that it is automatically configured for the advanced Web services features. For detailed instructions about using the Configuration Wizard to create and update WebLogic Server domains, see *Creating Domains Using the Configuration Wizard*.

To create a domain that is automatically configured for the advanced Web service features:

1. Start the Configuration Wizard.
2. In the Welcome window, select **Create a new WebLogic domain**.
3. Click **Next**.

4. Select **Generate a domain configured automatically to support the following products** and select **WebLogic Advanced Web Services for JAX-RPC Extension**.
5. Click **Next**.
6. Enter the name and location of the domain and click **Next**.
7. Configure the administrator user name and password and click **Next**.
8. Configure the server start mode and JDK and click **Next**.
9. If you want to further configure the JMS services, file stores, or any other feature, select the items on the Select Optional Configuration screen. This is not typical.

   Otherwise, leave all items deselected and click **Next**.
10. When you reach the Configuration Summary screen, verify the domain details and click **Create**.
11. Click **Done** to exit.

To extend an existing domain so that it is automatically configured for these Web Services features:

1. Start the Configuration Wizard.
2. In the Welcome window, select **Extend an Existing WebLogic Domain**.
3. Click **Next**.
4. Select the domain to which you want to apply the extension template.
5. Click **Next**.
6. Select **Extend my domain automatically to support the following added products** and select **WebLogic Advanced Web Services for JAX-RPC Extension**.
7. Click **Next**.
8. If you want to further configure the JMS services or file stores, select the items on the Select Optional Configuration screen. This is not typical.

   Otherwise, leave all items deselected and click **Next**.
9. Verify that you are extending the correct domain, then click **Extend**.
10. Click **Done** to exit.

3.3 Developing WebLogic Web Services Starting From Java: Main Steps

This section describes the general procedure for developing WebLogic Web Services starting from Java—in effect, coding the JWS file from scratch and later generating the WSDL file that describes the service. See Chapter 2, "Examples for JAX-RPC Web Service Developers" for specific examples of this process.

The following procedure is just a recommendation; if you have set up your own development environment, you can use this procedure as a guide for updating your existing environment to develop WebLogic Web Services.

> **Note:** This procedure does not use the WebLogic Web Services split development directory environment. If you are using this development environment, and would like to integrate Web Services development into it, see Section 3.13, "Integrating Web Services Into the WebLogic Split Development Directory Environment" for details.

Table 3–1 Steps to Develop Web Services Starting From Java

#	Step	Description
1	Set up the environment.	Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) command, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`ORACLE_HOME`*`/user_projects/domains/`*`domainName`*, where *`ORACLE_HOME`* is the directory you specified as Oracle Home when you installed Oracle WebLogic Server and *`domainName`* is the name of your domain.
2	Create a project directory.	The project directory will contain the JWS file, Java source for any user-defined data types, and the Ant `build.xml` file. You can name the project directory anything you want.
3	Create the JWS file that implements the Web Service.	See Section 4.3, "Programming the JWS File: Typical Steps."
4	Create user-defined data types. (Optional)	If your Web Service uses user-defined data types, create the JavaBeans that describes them. See Section 4.6, "Programming the User-Defined Java Data Type."
5	Create a basic Ant build file, `build.xml`.	See Section 3.5, "Creating the Basic Ant build.xml File."
6	Run the `jwsc` Ant task against the JWS file.	The `jwsc` Ant task generates source code, data binding artifacts, deployment descriptors, and so on, into an output directory. The `jwsc` Ant task generates an Enterprise application directory structure at this output directory; later you deploy this exploded directory to WebLogic Server as part of the iterative development process. See Section 3.6, "Running the jwsc WebLogic Web Services Ant Task."
7	Deploy the Web Service to WebLogic Server.	See Section 3.9, "Deploying and Undeploying WebLogic Web Services."
8	Browse to the WSDL of the Web Service.	Browse to the WSDL of the Web Service to ensure that it was deployed correctly. See Section 3.10, "Browsing to the WSDL of the Web Service."
9	Test the Web Service.	See Section 3.12, "Testing the Web Service."
10	Edit the Web Service. (Optional)	To make changes to the Web Service, update the JWS file, undeploy the Web Service as described in Section 3.9, "Deploying and Undeploying WebLogic Web Services," then repeat the steps starting from running the `jwsc` Ant task (Step 6).

See Chapter 4, "Programming the JWS File" for information on writing client applications that invoke a Web Service.

3.4 Developing WebLogic Web Services Starting From a WSDL File: Main Steps

This section describes the general procedure for developing WebLogic Web Services based on an existing WSDL file. See Chapter 3, "Developing JAX-RPC Web Services" for a specific example of this process.

The procedure is just a recommendation; if you have set up your own development environment, you can use this procedure as a guide for updating your existing environment to develop WebLogic Web Services.

It is assumed in this procedure that you already have an existing WSDL file.

> **Note:** This procedure does not use the WebLogic Web Services split development directory environment. If you are using this development environment, and would like to integrate Web Services development into it, see Section 3.13, "Integrating Web Services Into the WebLogic Split Development Directory Environment" for details.

Table 3–2 Steps to Develop Web Services Starting From Java

#	Step	Description
1	Set up the environment.	Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) command, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`ORACLE_HOME`*`/user_projects/domains/`*`domainName`*, where *`ORACLE_HOME`* is the directory you specified as Oracle Home when you installed Oracle WebLogic Server and *`domainName`* is the name of your domain.
2	Create a project directory.	The project directory will contain the generated artifacts and the Ant `build.xml` file.
3	Create a basic Ant build file, `build.xml`.	See Section 3.5, "Creating the Basic Ant build.xml File."
4	Put your WSDL file in a directory that the `build.xml` Ant build file is able to read.	For example, you can put the WSDL file in a `wsdl_files` child directory of the project directory.
5	Run the `wsdlc` Ant task against the WSDL file.	The `wsdlc` Ant task generates the JWS service endpoint interface (SEI), the stubbed-out JWS class file, JavaBeans that represent the XML Schema data types, and so on, into output directories. See Section 3.7, "Running the wsdlc WebLogic Web Services Ant Task."
6	Update the stubbed-out JWS file generated by the `wsdlc` Ant task.	The `wsdlc` Ant task generates a stubbed-out JWS file. You need to add your business code to the Web Service so it behaves as you want. See Section 3.8, "Updating the Stubbed-out JWS Implementation Class File Generated By wsdlc."
7	Run the `jwsc` Ant task against the JWS file.	Specify the artifacts generated by the `wsdlc` Ant task as well as your updated JWS implementation file, to generate an Enterprise Application that implements the Web Service. See Section 3.6, "Running the jwsc WebLogic Web Services Ant Task."
8	Deploy the Web Service to WebLogic Server.	See Section 3.9, "Deploying and Undeploying WebLogic Web Services."

Table 3–2 (Cont.) Steps to Develop Web Services Starting From Java

#	Step	Description
9	Browse to the WSDL of the Web Service.	Browse to the WSDL of the Web Service to ensure that it was deployed correctly. See Section 3.10, "Browsing to the WSDL of the Web Service." The URL used to invoke the WSDL of the deployed Web Service is essentially the same as the value of the `location` attribute of the `<address>` element in the original WSDL (except for the host and port values which now correspond to the host and port of the WebLogic Server instance to which you deployed the service.) This is because the `wsdlc` Ant task generated values for the `contextPath` and `serviceURI` of the `@WLHttpTransport` annotation in the JWS implementation file so that together they create the same URI as the endpoint address specified in the original WSDL.
10	Test the Web Service.	See Section 3.12, "Testing the Web Service."
11	Edit the Web Service. (Optional)	To make changes to the Web Service, update the JWS file, undeploy the Web Service as described in Section 3.9, "Deploying and Undeploying WebLogic Web Services," then repeat the steps starting from running the `jwsc` Ant task (Step 6).

See Chapter 6, "Developing JAX-RPC Web Service Clients" for information on writing client applications that invoke a Web Service.

3.5 Creating the Basic Ant build.xml File

Ant uses build files written in XML (default name `build.xml`) that contain a `<project>` root element and one or more targets that specify different stages in the Web Services development process. Each target contains one or more tasks, or pieces of code that can be executed. This section describes how to create a basic Ant build file; later sections describe how to add targets to the build file that specify how to execute various stages of the Web Services development process, such as running the `jwsc` Ant task to process a JWS file and deploying the Web Service to WebLogic Server.

The following skeleton `build.xml` file specifies a default `all` target that calls all other targets that will be added in later sections:

```
<project default="all">
  <target name="all"
          depends="clean,build-service,deploy" />
  <target name="clean">
     <delete dir="output" />
   </target>
  <target name="build-service">
      <!--add jwsc and related tasks here -->
   </target>
  <target name="deploy">
       <!--add wldeploy task here -->
  </dftarget>
</project>
```

3.6 Running the jwsc WebLogic Web Services Ant Task

The `jwsc` Ant task takes as input a JWS file that contains JWS annotations and generates all the artifacts you need to create a WebLogic Web Service. The JWS file can be either one you coded yourself from scratch or one generated by the `wsdlc` Ant task. The `jwsc`-generated artifacts include:

- JSR-109 Web Service class file.
- All required deployment descriptors, including:
  - Standard and WebLogic-specific Web Services deployment descriptors: `webservices.xml` and `weblogic-webservices.xml`.
  - JAX-RPC mapping files.
  - Java class-implemented Web Services: `web.xml` and `weblogic.xml`.
  - EJB-implemented Web Services: `ejb-jar.xml` and `weblogic-ejb-jar.xml`.
  - Ear deployment descriptor files: `application.xml` and `weblogic-application.xml`.
- The XML Schema representation of any Java user-defined types used as parameters or return values to the Web Service operations.
- The WSDL file that publicly describes the Web Service.

If you are running the `jwsc` Ant task against a JWS file generated by the `wsdlc` Ant task, the `jwsc` task does not generate these artifacts, because the `wsdlc` Ant task already generated them for you and packaged them into a JAR file. In this case, you use an attribute of the `jwsc` Ant task to specify this `wsdlc`-generated JAR file.

After generating all the required artifacts, the `jwsc` Ant task compiles the Java files (including your JWS file), packages the compiled classes and generated artifacts into a deployable JAR archive file, and finally creates an exploded Enterprise Application directory that contains the JAR file.

You can use the `jwsc` Ant task to perform the following advanced tasks:

- Process multiple JWS files at once. You can choose to package each resulting Web Service into its own Web application WAR file, or group all of the Web Services into a single WAR file.
- Specify the transports (HTTP/HTTPS/JMS) that client applications can use when invoking the Web Service, possibly overriding any existing `@WLXXXTransport` annotations, as described in Chapter 3.6.1, "Specifying the Transport Used to Invoke the Web Service."
- Automatically generate the JAX-RPC client stubs of any other Web Service that is invoked within the JWS file.
- Update an existing Enterprise Application or Web application, rather than generate a completely new one.

To run the `jwsc` Ant task, add the following `taskdef` and `build-service` target to the `build.xml` file:

```
<taskdef name="jwsc"
          classname="weblogic.wsee.tools.anttasks.JwscTask" />
<target name="build-service">
    <jwsc
       srcdir="src_directory"
       destdir="ear_directory"
```

```
        >
        <jws file="JWS_file"
             compiledWsdl="WSDLC_Generated_JAR"
              type="WebService_type"/>
    </jwsc>
</target>
```

where:

- `ear_directory` refers to an Enterprise Application directory that will contain all the generated artifacts.
- `src_directory` refers to the top-level directory that contains subdirectories that correspond to the package name of your JWS file.
- `JWS_file` refers to the full pathname of your JWS file, relative to the value of the `src_directory` attribute.
- *WSDLC_Generated_JAR* refers to the JAR file generated by the `wsdlc` Ant task that contains the JWS SEI and data binding artifacts that correspond to an existing WSDL file.

> **Note:** You specify this attribute only in the "starting from WSDL" use case; this procedure is described in Section 3.4, "Developing WebLogic Web Services Starting From a WSDL File: Main Steps."

- *`WebService_type`* specifies the type of Web Service. This value can be set to JAXWS or JAXRPC.

The required `taskdef` element specifies the full class name of the `jwsc` Ant task.

Only the `srcdir` and `destdir` attributes of the `jwsc` Ant task are required. This means that, by default, it is assumed that Java files referenced by the JWS file (such as JavaBeans input parameters or user-defined exceptions) are in the same package as the JWS file. If this is not the case, use the `sourcepath` attribute to specify the top-level directory of these other Java files.

See "jwsc" in the *WebLogic Web Services Reference for Oracle WebLogic Server* for complete documentation and examples about the `jwsc` Ant task.

3.6.1 Specifying the Transport Used to Invoke the Web Service

The `<jws>` child element of `jwsc` includes the following optional child elements for specifying the transports (HTTP/S or JMS) that are used to invoke the Web service:

- `WLHttpTransport`—Specifies the context path and service URI sections of the URL used to invoke the Web service over the HTTP/S transport, as well as the name of the port in the generated WSDL.
- `WLJMSTransport`—Specifies the context path and service URI sections of the URL used to invoke the Web service over the JMS transport, as well as the name of the port in the generated WSDL. You also specify the name of the JMS queue and connection factory that you have already configured for JMS transport.

The following guidelines describe the usage of the transport elements for the `jwsc` Ant task:

- The transports you specify to `jwsc` *always override any corresponding transport annotations in the JWS file. In addition, all* attributes of the transport annotation are ignored, even if you have not explicitly specified the corresponding attribute for

the transport element, in which case the default value of the transport element attribute is used.

- You can specify both transport elements for a particular JWS file. However, you can specify only *one* instance of a particular transport element. For example, although you cannot specify two different `<WLHttpTransport>` elements for a given JWS file, you can specify one `<WLHttpTransport>` and one `<WLJmsTransport>` element.
- The value of the `serviceURI` attribute can be the same when you specify both `<WLJMSTransport>` and `<WLHttpTransport>`.
- All transports associated with a particular JWS file must specify the *same* `contextPath` attribute value.
- If you specify more than one transport element for a particular JWS file, the value of the `portName` attribute for each element must be unique among all elements. This means that you must explicitly specify this attribute if you add more than one transport child element to `<jws>`, because the default value of the element will always be the same and thus cause an error when running the `jwsc` Ant task.
- If you do not specify any transport as either one of the transport elements to the `jwsc` Ant task or a transport annotation in the JWS file, then the Web service's default URL corresponds to the default value of the `WLHttpTransport` element.

For JAX-RPC Web services, when you program your JWS file, you can use an annotation to specify the transport that clients use to invoke the Web service, in particular `@weblogic.jws.WLHttpTransport` or `@weblogic.jws.WLJMSTransport`. You can specify only *one* instance of a particular transport annotation in the JWS file. For example, although you cannot specify two different `@WLHttpTransport` annotations, you can specify one `@WLHttpTransport` and one `@WLJmsTransport` annotation. However, you might not know at the time that you are coding the JWS file which transports best suits your needs. For this reason, it is often better to specify the transport at build-time.

3.6.2 Defining the Context Path of a WebLogic Web Service

There are a variety of places where the context path (also called context root) of a WebLogic Web service can be specified. This section describes how to determine which is the true context path of the service based on its configuration, even if it is has been set in multiple places.

In the context of this discussion, a Web service context path is the string that comes after the `host:port` portion of the Web service URL. For example, if the deployed WSDL of a WebLogic Web service is as follows:

```
http://hostname:7001/financial/GetQuote?WSDL
```

The context path for this Web service is `financial`.

The following list describes the order of precedence, from most to least important, of all possible context path specifications:

1. The `contextPath` attribute of the `<module>` element and `<jws>` element (when used as a direct child of the `jwsc` Ant task.)
2. The `contextPath` attribute of the `<WL`*`XXX`*`Transport>` child elements of `<jws>`.
3. The `contextPath` attribute of the `@WL`*`XXX`*`Transport` JWS annotations, as described in Section 4.3.4, "Specifying the Context Path and Service URI of the Web Service (@WLHttpTransport Annotation)."

4. The default value of the context path, which is the name of the JWS file without any extension.

Suppose, for example, that you specified the `@WLHttpTransport` annotation in your JAX-RPC JWS file and set its `contextPath` attribute to `financial`. If you do not specify any additional `contextPath` attributes in the `jwsc` Ant task in your `build.xml` file, then the context path for this Web service would be `financial`.

Assume that you then update the `build.xml` file and add a `<WLHttpTransport>` child element to the `<jws>` element that specifies the JWS file and set its `contextPath` attribute to `finance`. The context path of the Web service would now be `finance`. If, however, you then group the `<jws>` element (including its child `<WLHttpTransport>` element) under a `<module>` element, and set its `contextPath` attribute to `money`, then the context path of the Web service would now be `money`.

If you do not specify *any* `contextPath` attribute in either the JWS file or the `jwsc` Ant task, then the context path of the Web service is the default value: the name of the JWS file without its `*.java` extension.

If you group two or more `<jws>` elements under a `<module>` element and do not set the context path using any of the other options listed above, then you *must* specify the `contextPath` attribute of `<module>` to specify the common context path used by all the Web services in the module. Otherwise, the default context paths for all the Web services in the module are going to be different (due to different names of the implementing JWS files), which is not allowed in a single WAR file.

3.6.3 Examples of Using jwsc

The following `build.xml` excerpt shows a basic example of running the `jwsc` Ant task on a JWS file:

```
<taskdef name="jwsc"
          classname="weblogic.wsee.tools.anttasks.JwscTask" />
<target name="build-service">
    <jwsc
       srcdir="src"
       destdir="output/helloWorldEar">
       <jws
          file="examples/webservices/hello_world/HelloWorldImpl.java"
          type="JAXRPC"/>
    </jwsc>
</target>
```

In the example:

- The Enterprise application will be generated, in exploded form, in `output/helloWorldEar`, relative to the current directory.
- The JWS file is called `HelloWorldImpl.java`, and is located in the `src/examples/webservices/hello_world` directory, relative to the current directory. This implies that the JWS file is in the package `examples.webservices.helloWorld`.
- A JAX-RPC Web Service is generated.

The following example is similar to the preceding one, except that it uses the `compiledWsdl` attribute to specify the JAR file that contains `wsdlc`-generated artifacts (for the "starting with WSDL" use case):

```
<taskdef name="jwsc"
```

```
         classname="weblogic.wsee.tools.anttasks.JwscTask" />
<target name="build-service">
  <jwsc
     srcdir="src"
     destdir="output/wsdlcEar">
    <jws
        file="examples/webservices/wsdlc/TemperaturePortTypeImpl.java"
         compiledWsdl="output/compiledWsdl/TemperatureService_wsdl.jar"
         type="JAXRPC"/>
  </jwsc>
</target>
```

In the preceding example, the `TemperaturePortTypeImpl.java` file is the stubbed-out JWS file that you updated to include your business logic. Because the `compiledWsdl` attribute is specified and points to a JAR file, the `jwsc` Ant task does not regenerate the artifacts that are included in the JAR.

To actually run this task, type at the command line the following:

```
prompt> ant build-service
```

3.7 Running the wsdlc WebLogic Web Services Ant Task

The `wsdlc` Ant task takes as input a WSDL file and generates artifacts that together partially implement a WebLogic Web Service. These artifacts include:

- JWS service endpoint interface (SEI) that implements the Web Service described by the WSDL file.
- JWS implementation file that contains a partial (stubbed-out) implementation of the generated JWS SEI. This file must be customized by the developer.
- Data binding artifacts used by WebLogic Server to convert between the XML and Java representations of the Web Service parameters and return values.
- Optional Javadocs for the generated JWS SEI.

The `wsdlc` Ant task packages the JWS SEI and data binding artifacts together into a JAR file that you later specify to the `jwsc` Ant task. You never need to update this JAR file; the only file you update is the JWS implementation class.

To run the `wsdlc` Ant task, add the following `taskdef` and `generate-from-wsdl` targets to the `build.xml` file:

```
<taskdef name="wsdlc"
         classname="weblogic.wsee.tools.anttasks.WsdlcTask"/>
<target name="generate-from-wsdl">
  <wsdlc
       srcWsdl="WSDL_file"
       destJwsDir="JWS_interface_directory"
       destImplDir="JWS_implementation_directory"
       packageName="Package_name"
       type="WebService_type"/>
 </target>
```

where:

- *`WSDL_file`* refers to the name of the WSDL file from which you want to generate a partial implementation, including its absolute or relative pathname.

- *JWS_interface_directory* refers to the directory into which the JAR file that contains the JWS SEI and data binding artifacts should be generated.

  The name of the generated JAR file is `WSDLFile_wsdl.jar`, where `WSDLFile` refers to the root name of the WSDL file. For example, if the name of the WSDL file you specify to the file attribute is `MyService.wsdl`, then the generated JAR file is `MyService_wsdl.jar`.

- *JWS_implementation_directory* refers to the top directory into which the stubbed-out JWS implementation file is generated. The file is generated into a subdirectory hierarchy corresponding to its package name.

  The name of the generated JWS file is `PortTypeImpl.java`, where `PortType` refers to the `name` attribute of the `<portType>` element in the WSDL file for which you are generating a Web Service. For example, if the port type name is `MyServicePortType`, then the JWS implementation file is called `MyServicePortTypeImpl.java`.

- *Package_name* refers to the package into which the generated JWS SEI and implementation files should be generated. If you do not specify this attribute, the `wsdlc` Ant task generates a package name based on the `targetNamespace` of the WSDL.

- *WebService_type* specifies the type of Web Service. This value can be set to JAXWS or JAXRPC.

The required `taskdef` element specifies the full class name of the `wsdlc` Ant task.

Only the `srcWsdl` and `destJwsDir` attributes of the `wsdlc` Ant task are required. Typically, however, you generate the stubbed-out JWS file to make your programming easier. Oracle recommends you explicitly specify the package name in case the `targetNamespace` of the WSDL file is not suitable to be converted into a readable package name.

The following `build.xml` excerpt shows an example of running the `wsdlc` Ant task against a WSDL file:

```
<taskdef name="wsdlc"
         classname="weblogic.wsee.tools.anttasks.WsdlcTask"/>
<target name="generate-from-wsdl">
    <wsdlc
         srcWsdl="wsdl_files/TemperatureService.wsdl"
         destJwsDir="output/compiledWsdl"
         destImplDir="impl_output"
         packageName="examples.webservices.wsdlc"
         type="JAXRPC" />
</target>
```

In the example:

- The existing WSDL file is called `TemperatureService.wsdl` and is located in the `wsdl_files` subdirectory of the directory that contains the `build.xml` file.
- The JAR file that will contain the JWS SEI and data binding artifacts is generated to the `output/compiledWsdl` directory; the name of the JAR file is `TemperatureService_wsdl.jar`.
- The package name of the generated JWS files is `examples.webservices.wsdld`.
- The stubbed-out JWS file is generated into the `impl_output/examples/webservices/wsdlc` directory relative to the current directory.

- Assuming that the port type name in the WSDL file is `TemperaturePortType`, then the name of the JWS implementation file is `TemperaturePortTypeImpl.java`.
- A JAX-RPC Web Service is generated.

To actually run this task, type the following at the command line:

```
prompt> ant generate-from-wsdl
```

See "wsdlc" in *WebLogic Web Services Reference for Oracle WebLogic Server* for more information.

3.8 Updating the Stubbed-out JWS Implementation Class File Generated By wsdlc

The `wsdlc` Ant task generates the stubbed-out JWS implementation file into the directory specified by its `destImplDir` attribute; the name of the file is `PortTypeImpl.java`, where `PortType` is the name of the portType in the original WSDL. The class file includes everything you need to compile it into a Web Service, except for your own business logic.

The JWS class implements the JWS Web Service endpoint interface that corresponds to the WSDL file; the JWS SEI is also generated by `wsdlc` and is located in the JAR file that contains other artifacts, such as the Java representations of XML Schema data types in the WSDL and so on. The public methods of the JWS class correspond to the operations in the WSDL file.

The `wsdlc` Ant task automatically includes the `@WebService` and `@WLHttpTransport` annotations in the JWS implementation class; the values of the attributes corresponds to the equivalent values in the WSDL. For example, the `serviceName` attribute of `@WebService` is the same as the `name` attribute of the `<service>` element in the WSDL file; the `contextPath` and `serviceUri` attributes of `@WLHttpTransport` together make up the endpoint address specified by the `location` attribute of the `<address>` element in the WSDL.

When you update the JWS file, you add Java code to the methods so that the corresponding Web Service operations operate as required. Typically, the generated JWS file contains comments where you should add code, such as:

```
//replace with your impl here
```

In addition, you can add additional JWS annotations to the file, with the following restrictions:

- You can include the following annotations from the standard (JSR-181) `javax.jws` package in the JWS implementation file: `@WebService`, `@HandlerChain`, `@SOAPMessageHandler`, and `@SOAPMessageHandlers`. If you specify any other JWS annotation from the `javax.jws` package, the `jwsc` Ant task returns error when you try to compile the JWS file into a Web Service.
- You can specify *only* the `serviceName`, `endpointInterface`, and `targetNamespace` attributes of the `@WebService` annotation. Use the `serviceName` attribute to specify a different `<service>` WSDL element from the one that the `wsdlc` Ant task used, in the rare case that the WSDL file contains more than one `<service>` element. Use the `endpointInterface` attribute to specify the JWS SEI generated by the `wsdlc` Ant task. Use the `targetNamespace` attribute to specify the namespace of a WSDL service, which can be different from the on in JWS SEI.

- You can specify WebLogic-specific JWS annotations, as required.

After you have updated the JWS file, Oracle recommends that you move it to an official source location, rather than leaving it in the `wsdlc` output directory.

The following example shows the `wsdlc`-generated JWS implementation file from the WSDL shown in Section 2.3.1, "Sample WSDL File"; the text in **bold** indicates where you would add Java code to implement the single operation (`getTemp`) of the Web Service:

```
package examples.webservices.wsdlc;
import javax.jws.WebService;
 import weblogic.jws.*;
/**
  * TemperaturePortTypeImpl class implements web service endpoint interface
  * TemperaturePortType */
@WebService(
   serviceName="TemperatureService",
   endpointInterface="examples.webservices.wsdlc.TemperaturePortType")
@WLHttpTransport(
   contextPath="temp",
   serviceUri="TemperatureService",
   portName="TemperaturePort")
public class TemperaturePortTypeImpl implements TemperaturePortType {
  public TemperaturePortTypeImpl() {
  }
  public float getTemp(java.lang.String zipcode)
  {
    //replace with your impl here
     return 0;
  }
}
```

3.9 Deploying and Undeploying WebLogic Web Services

Because Web Services are packaged as Enterprise Applications, deploying a Web Service simply means deploying the corresponding EAR file or exploded directory.

There are a variety of ways to deploy WebLogic applications, from using the Administration Console to using the `weblogic.Deployer` Java utility. There are also various issues you must consider when deploying an application to a production environment as opposed to a development environment. For a complete discussion about deployment, see *Deploying Applications to Oracle WebLogic Server*.

This guide, because of its development nature, discusses just two ways of deploying Web Services:

- Section 3.9.1, "Using the wldeploy Ant Task to Deploy Web Services"
- Section 3.9.2, "Using the Administration Console to Deploy Web Services"

3.9.1 Using the wldeploy Ant Task to Deploy Web Services

The easiest way to deploy a Web Service as part of the iterative development process is to add a target that executes the `wldeploy` WebLogic Ant task to the same `build.xml` file that contains the `jwsc` Ant task. You can add tasks to both deploy and undeploy the Web Service so that as you add more Java code and regenerate the service, you can redeploy and test it iteratively.

To use the `wldeploy` Ant task, add the following target to your `build.xml` file:

```
<target name="deploy">
    <wldeploy action="deploy"
       name="DeploymentName"
       source="Source" user="AdminUser"
       password="AdminPassword"
       adminurl="AdminServerURL"
       targets="ServerName"/>
</target>
```

where:

- *DeploymentName* refers to the deployment name of the Enterprise Application, or the name that appears in the Administration Console under the list of deployments.
- *Source* refers to the name of the Enterprise Application EAR file or exploded directory that is being deployed. By default, the `jwsc` Ant task generates an exploded Enterprise Application directory.
- *AdminUser* refers to administrative username.
- *AdminPassword* refers to the administrative password.
- *AdminServerURL* refers to the URL of the Administration Server, typically `t3://localhost:7001`.
- *ServerName* refers to the name of the WebLogic Server instance to which you are deploying the Web Service.

For example, the following `wldeploy` task specifies that the Enterprise Application exploded directory, located in the `output/ComplexServiceEar` directory relative to the current directory, be deployed to the `myServer` WebLogic Server instance. Its deployed name is `ComplexServiceEar`.

```
<target name="deploy">
  <wldeploy action="deploy"
     name="ComplexServiceEar"
     source="output/ComplexServiceEar" user="weblogic"
     password="weblogic" verbose="true"
     adminurl="t3://localhost:7001"
     targets="myserver"/>
</target>
```

To actually deploy the Web Service, execute the `deploy` target at the command-line:

```
prompt> ant deploy
```

You can also add a target to easily undeploy the Web Service so that you can make changes to its source code, then redeploy it:

```
<target name="undeploy">
  <wldeploy action="undeploy"
     name="ComplexServiceEar"
     user="weblogic"
     password="weblogic" verbose="true"
     adminurl="t3://localhost:7001"
     targets="myserver"/>
</target>
```

When undeploying a Web Service, you do not specify the `source` attribute, but rather undeploy it by its name.

3.9.2 Using the Administration Console to Deploy Web Services

To use the Administration Console to deploy the Web Service, first invoke it in your browser using the following URL:

```
http://host:port/console
```

where:

- *host* refers to the computer on which WebLogic Server is running.
- *port* refers to the port number on which WebLogic Server is listening (default value is `7001`).

Then use the deployment assistants to help you deploy the Enterprise application. For more information on the Administration Console, see the *Oracle WebLogic Server Administration Console Online Help*.

3.10 Browsing to the WSDL of the Web Service

You can display the WSDL of the Web Service in your browser to ensure that it has deployed correctly.

The following URL shows how to display the Web Service WSDL in your browser:

```
http://host:port/contextPath/serviceUri?WSDL
```

where:

- *host* refers to the computer on which WebLogic Server is running (for example, `localhost`).
- *port* refers to the port number on which WebLogic Server is listening (default value is `7001`).
- *contextPath* refers to the context root of the Web Service. There are many places to set the context root (the `contextPath` attribute of the `@WLHttpTransport` annotation, the `<WLHttpTransport>`, `<module>`, or `<jws>` element of `jwsc`) and certain methods take precedence over others. See Chapter 3.6.2, "Defining the Context Path of a WebLogic Web Service."
- `serviceUri` refers to the value of the `serviceUri` attribute of the `@WLHttpTransport` JWS annotation of the JWS file that implements your Web Service or `<WLHttpTransport>` child element of the `jwsc` Ant task; the second takes precedence over the first. If you do not specify *any* `serviceUri` attribute in either the JWS file or the `jwsc` Ant task, then the `serviceUri` of the Web Service is the default value: the name of the JWS file without its `*.java` extension.

For example, assume you specified the following `@WLHttpTransport` annotation in the JWS file that implements your Web Service

```
...
@WLHttpTransport(contextPath="complex",
                 serviceUri="ComplexService",
                 portName="ComplexServicePort")
/**
  * This JWS file forms the basis of a WebLogic Web Service.
  *
  */
public class ComplexServiceImpl {
...
```

Further assume that you do *not* override the `contextPath` or `serviceURI` values by setting equivalent attributes for the `<WLHttpTransport>` element of the `jwsc` Ant task. Then the URL to view the WSDL of the Web Service, assuming the service is running on a host called `ariel` at the default port number (`7001`), is:

```
http://ariel:7001/complex/ComplexService?WSDL
```

3.11 Configuring the Server Address Specified in the Dynamic WSDL

The WSDL of a deployed Web Service (also called *dynamic WSDL*) includes an `<address>` element that assigns an address (URI) to a particular Web Service port. For example, assume that the following WSDL snippet partially describes a deployed WebLogic Web Service called `ComplexService`:

```
<definitions name="ComplexServiceDefinitions"
             targetNamespace="http://example.org">
...
  <service name="ComplexService">
     <port binding="s0:ComplexServiceSoapBinding" name="ComplexServicePort">
       <s1:address location="http://myhost:7101/complex/ComplexService"/>
     </port>
   </service>
</definitions>
```

The preceding example shows that the `ComplexService` Web Service includes a port called `ComplexServicePort`, and this port has an address of `http://myhost:7101/complex/ComplexService`.

WebLogic Server determines the `complex/ComplexService` section of this address by examining the `contextPath` and `serviceURI` attributes of the `@WLXXXTransport` annotations or `jwsc` elements, as described in Section 3.10, "Browsing to the WSDL of the Web Service." However, the method WebLogic Server uses to determine the protocol and host section of the address (`http://myhost:7101`, in the example) is more complicated, as described below. For clarity, this section uses the term *server address* to refer to the protocol and host section of the address.

The server address that WebLogic Server publishes in a dynamic WSDL of a deployed Web Service depends on whether the Web Service can be invoked using HTTP/S or JMS, whether you have configured a proxy server, whether the Web Service is deployed to a cluster, or whether the Web Service is actually a callback service.

The following sections reflect these different configuration options, and provide links to procedural information about changing the configuration to suit your needs.

- Section 3.11.1, "Web Service is not a callback service and can be invoked using HTTP/S"
- Section 3.11.2, "Web Service is not a callback service and can be invoked using JMS Transport"
- Section 3.11.3, "Web Service is a callback service"
- Section 3.11.4, "Web Service is invoked using a proxy server"

It is assumed in the sections that you use the WebLogic Server Administration Console to configure cluster and standalone servers.

3.11.1 Web Service is not a callback service and can be invoked using HTTP/S

1. If the Web Service is deployed to a cluster, and the cluster `Frontend Host`, `Frontend HTTP Port`, and `Frontend HTTPS Port` are set, then WebLogic Server uses these values in the server address of the dynamic WSDL.

   See "Configure HTTP Settings for a Cluster" in the *Oracle WebLogic Server Administration Console Online Help*.

2. If the preceding cluster values are not set, but the `Frontend Host`, `Frontend HTTP Port`, and `Frontend HTTPS Port` values are set for the *individual server* to which the Web Service is deployed, then WebLogic Server uses these values in the server address.

   See "Configure HTTP Protocol" in the *Oracle WebLogic Server Administration Console Online Help*.

3. If these values are not set for the cluster or individual server, then WebLogic Server uses the server address of the WSDL request in the dynamic WSDL.

3.11.2 Web Service is not a callback service and can be invoked using JMS Transport

1. If the Web Service is deployed to a cluster and the `Cluster Address` is set, then WebLogic Server uses this value in the server address of the dynamic WSDL.

   See "Configure Clusters" in the *Oracle WebLogic Server Administration Console Online Help*.

2. If the cluster address is not set, or the Web Service is deployed to a standalone server, and the `Listen Address` of the server to which the Web Service is deployed is set, then WebLogic Server uses this value in the server address.

   See "Configure Listen Addresses" in the *Oracle WebLogic Server Administration Console Online Help*.

3.11.3 Web Service is a callback service

1. If the callback service is deployed to a cluster, and the cluster `Frontend Host`, `Frontend HTTP Port`, and `Frontend HTTPS Port` are set, then WebLogic Server uses these values in the server address of the dynamic WSDL.

   See "Configure HTTP Settings for a Cluster" in the *Oracle WebLogic Server Administration Console Online Help*.

2. If the callback service is deployed to either a cluster or a standalone server, and the preceding cluster values are not set, but the `Frontend Host`, `Frontend HTTP Port`, and `Frontend HTTPS Port` values are set for the *individual server* to which the callback service is deployed, then WebLogic Server uses these values in the server address.

   See "Configure HTTP Protocol" in the *Oracle WebLogic Server Administration Console Online Help*.

3. If the callback service is deployed to a cluster, but none of the preceding values are set, but the `Cluster Address` is set, then WebLogic Server uses this value in the server address.

   See "Configure Clusters" in the *Oracle WebLogic Server Administration Console Online Help*.

4. If none of the preceding values are set, but the `Listen Address` of the server to which the callback service is deployed is set, then WebLogic Server uses this value in the server address.

   See "Configure Listen Addresses" in the *Oracle WebLogic Server Administration Console Online Help*.

3.11.4 Web Service is invoked using a proxy server

Although not required, Oracle recommends that you explicitly set the `Frontend Host`, `FrontEnd HTTP Port`, and `Frontend HTTPS Port` of either the cluster or individual server to which the Web Service is deployed to point to the proxy server.

See "Configure HTTP Settings for a Cluster" or "Configure HTTP Protocol" in the *Oracle WebLogic Server Administration Console Online Help*.

3.12 Testing the Web Service

After you have deployed a WebLogic Web Service, you can use the Web Services Test Client, included in the WebLogic Administration Console, to test your service without writing code. You can quickly and easily test any Web Service, including those with complex types and those using advanced features of WebLogic Server such as conversations. The test client automatically maintains a full log of requests allowing you to return to the previous call to view the results.

To test a deployed Web Service using the Administration Console, follow these steps:

1. Invoke the Administration Console in your browser using the following URL:

   `http://`*`host`*`:`*`port`*`/console`

   where:

   - *`host`* refers to the computer on which WebLogic Server is running.
   - *`port`* refers to the port number on which WebLogic Server is listening (default value is `7001`).

2. Follow the procedure described in "Test a Web Service" in the *Oracle WebLogic Server Administration Console Online Help*.

3.13 Integrating Web Services Into the WebLogic Split Development Directory Environment

This section describes how to integrate Web Services development into the WebLogic split development directory environment. It is assumed that you understand this WebLogic feature and have set up this type of environment for developing standard Java Platform, Enterprise Edition (Java EE) Version 5 applications and modules, such as EJBs and Web applications, and you want to update the single `build.xml` file to include Web Services development.

For detailed information about the WebLogic split development directory environment, see "Creating a Split Development Directory for an Application" in *Developing Applications for Oracle WebLogic Server* and the `splitdir/helloWorldEar` example optionally installed with WebLogic Server, located in the *`EXAMPLES_HOME`*`/wl_server/examples/src/examples` directory, where *`EXAMPLES_HOME`* represents the directory in which the WebLogic Server code examples are configured. The default path is *`ORACLE_HOME`*`\user_projects\applications`. For more

information about the WebLogic Server code examples, see "Sample Applications and Code Examples" in *Understanding Oracle WebLogic Server*.

1. In the main project directory, create a directory that will contain the JWS file that implements your Web Service.

   For example, if your main project directory is called `/src/helloWorldEar`, then create a directory called `/src/helloWorldEar/helloWebService`:

   ```
   prompt> mkdir /src/helloWorldEar/helloWebService
   ```

2. Create a directory hierarchy under the `helloWebService` directory that corresponds to the package name of your JWS file.

   For example, if your JWS file is in the package `examples.splitdir.hello` package, then create a directory hierarchy `examples/splitdir/hello`:

   ```
   prompt> cd /src/helloWorldEar/helloWebService
    prompt> mkdir examples/splitdir/hello
   ```

3. Put your JWS file in the just-created Web Service subdirectory of your main project directory (`/src/helloWorldEar/helloWebService/examples/splitdir/hello` in this example.)

4. In the `build.xml` file that builds the Enterprise application, create a new target to build the Web Service, adding a call to the `jwsc` WebLogic Web Service Ant task, as described in Section 3.6, "Running the jwsc WebLogic Web Services Ant Task."

   The `jwsc srcdir` attribute should point to the top-level directory that contains the JWS file (`helloWebService` in this example). The `jwsc destdir` attribute should point to the same destination directory you specify for `wlcompile`, as shown in the following example:

   ```
   <target name="build.helloWebService">
     <jwsc
          srcdir="helloWebService"
          destdir="destination_dir"
          keepGenerated="yes" >
         <jws file="examples/splitdir/hello/HelloWorldImpl.java"
           type="JAXRPC" />
     </jwsc>
   </target>
   ```

   In the example, `destination_dir` refers to the destination directory that the other split development directory environment Ant tasks, such as `wlappc` and `wlcompile`, also use.

5. Update the main build target of the `build.xml` file to call the Web Service-related targets:

   ```
   <!-- Builds the entire helloWorldEar application -->
   <target name="build"
      description="Compiles helloWorldEar application and runs appc"
      depends="build-helloWebService,compile,appc" />
   ```

> **Note:** When you actually build your Enterprise Application, be sure you run the `jwsc` Ant task *before* you run the `wlappc` Ant task. This is because `wlappc` requires some of the artifacts generated by `jwsc` for it to execute successfully. In the example, this means that you should specify the `build-helloWebService` target *before* the `appc` target.

6. If you use the `wlcompile` and `wlappc` Ant tasks to compile and validate the entire Enterprise Application, be sure to exclude the Web Service source directory for both Ant tasks. This is because the `jwsc` Ant task already took care of compiling and packaging the Web Service. For example:

```
<target name="compile">
   <wlcompile srcdir="${src.dir}" destdir="${dest.dir}"
            excludes="appStartup,helloWebService">
    ...
    </wlcomplile>
...
 </target>
<target name="appc">
   <wlappc source="${dest.dir}" deprecation="yes" debug="false"
            excludes="helloWebService"/>
</target>
```

7. Update the `application.xml` file in the `META-INF` project source directory, adding a `<web>` module and specifying the name of the WAR file generated by the `jwsc` Ant task.

   For example, add the following to the `application.xml` file for the helloWorld Web Service:

```
<application>
...
  <module>
     <web>
       <web-uri>examples/splitdir/hello/HelloWorldImpl.war</web-uri>
       <context-root>/hello</context-root>
     </web>
   </module>
...
</application>
```

> **Note:** The `jwsc` Ant task always generates a Web Application WAR file from the JWS file that implements your Web Service, unless your JWS file explicitly implements `javax.ejb.SessionBean`. In that case you must add an `<ejb>` module element to the `application.xml` file instead.

Your split development directory environment is now updated to include Web Service development. When you rebuild and deploy the entire Enterprise Application, the Web Service will also be deployed as part of the EAR. You invoke the Web Service in the standard way described in Section 3.10, "Browsing to the WSDL of the Web Service."

4 Programming the JWS File

This chapter describes how to program the JWS file that implements the WebLogic Web services using Java API for XML-based RPC (JAX-RPC).

This chapter includes the following topics:

4.1 Overview of JWS Files and JWS Annotations

There are two ways to program a WebLogic Web service from scratch:

1. Annotate a standard EJB or Java class with Web service Java annotations, as defined by JSR-181, the JAX-WS specification, and by the WebLogic Web services programming model.
2. Combine a standard EJB or Java class with the various XML descriptor files and artifacts specified by JSR-109 (such as, deployment descriptors, WSDL files, data mapping descriptors, data binding artifacts for user-defined data types, and so on).

Oracle strongly recommends using option 1 above. Instead of authoring XML metadata descriptors yourself, the WebLogic Ant tasks and run time will generate the required descriptors and artifacts based on the annotations you include in your JWS. Not only is this process much easier, but it keeps the information about your Web service in a central location, the JWS file, rather than scattering it across many Java and XML files.

The Java Web Service (JWS) annotated file is the core of your Web service. It contains the Java code that determines how your Web service behaves. A JWS file is an ordinary Java class file that uses Java metadata annotations to specify the shape and

characteristics of the Web service. The JWS annotations you can use in a JWS file include the standard ones defined by the *Web Services Metadata for the Java Platform* specification (JSR-181), described at `http://www.jcp.org/en/jsr/detail?id=181`, plus a set of additional annotations based on the type of Web service you are building—JAX-WS or JAX-RPC. For a complete list of JWS annotations that are supported for JAX-WS and JAX-RPC Web services, see "Web Service Annotation Support" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

When programming the JWS file, you include annotations to program basic Web service features. The annotations are used at different levels, or targets, in your JWS file. Some are used at the class-level to indicate that the annotation applies to the entire JWS file. Others are used at the method-level and yet others at the parameter level.

4.2 Java Requirements for a JWS File

When you program your JWS file, you must follow a set of requirements, as specified by the *Web Services Metadata for the Java Platform* specification (JSR-181) at `http://www.jcp.org/en/jsr/detail?id=181`. In particular, the Java class that implements the Web service:

- Must be an outer public class, must not be declared `final`, and must not be `abstract`.
- Must have a default public constructor.
- Must not define a `finalize()` method.
- Must include, at a minimum, a `@WebService` JWS annotation at the class level to indicate that the JWS file implements a Web service.
- May reference a service endpoint interface by using the `@WebService.endpointInterface` annotation. In this case, it is assumed that the service endpoint interface exists and you cannot specify any other JWS annotations in the JWS file other than `@WebService.endpointInterface`, `@WebService.serviceName` and `@WebService.targetNamespace`.
- If JWS file does not implement a service endpoint interface, all public methods other than those inherited from `java.lang.Object` will be exposed as Web service operations. This behavior can be overridden by using the `@WebMethod` annotation to specify explicitly the public methods that are to be exposed. If a `@WebMethod` annotation is present, only the methods to which it is applied are exposed.

4.3 Programming the JWS File: Typical Steps

The following procedure describes the typical steps for programming a JWS file that implements a Web service.

> **Note:** It is assumed that you have created a JWS file and now want to add JWS annotations to it.

For more information about each of the JWS annotations, see "JWS Annotation Reference" in *WebLogic Web Services Reference for Oracle WebLogic Server*. See *Programming Advanced Features of JAX-RPC Web Services for Oracle WebLogic Server* for information on using other JWS annotations to program more advanced features, such as Web service reliable messaging, conversations, SOAP message handlers, and so on.

Table 4–1 Steps to Program the JWS File

#	Step	Description

1	Import the standard JWS annotations that will be used in your JWS file.	The standard JWS annotations are in either the `javax.jws` or `javax.jws.soap` package. For example:
		`import javax.jws.WebMethod;`
		`import javax.jws.WebService;`
		`import javax.jws.soap.SOAPBinding;`

2	Import the WebLogic-specific annotations used in your JWS file.	The WebLogic-specific annotations are in the `weblogic.jws` package. For example: `import weblogic.jws.WLHttpTransport;`
3	Add the standard required `@WebService` JWS annotation at the class level to specify that the Java class exposes a Web service.	See Section 4.3.2, "Specifying that the JWS File Implements a Web Service (@WebService Annotation)."
4	Add the standard `@SOAPBinding` JWS annotation at the class level to specify the mapping between the Web service and the SOAP message protocol. (Optional)	In particular, use this annotation to specify whether the Web service is document-literal, RPC-encoded, and so on. See Section 4.3.3, "Specifying the Mapping of the Web Service to the SOAP Message Protocol (@SOAPBinding Annotation)." Although this JWS annotation is not required, Oracle recommends you explicitly specify it in your JWS file to clarify the type of SOAP bindings a client application uses to invoke the Web service.
5	Add the WebLogic-specific `@WLHttpTransport` JWS annotation at the class level to specify the context path and service URI used in the URL that invokes the Web service. (Optional)	See Section 4.3.4, "Specifying the Context Path and Service URI of the Web Service (@WLHttpTransport Annotation)." Although this JWS annotation is not required, Oracle recommends you explicitly specify it in your JWS file so that it is clear what URL a client application uses to invoke the Web service.
6	Add the standard `@WebMethod` annotation for each method in the JWS file that you want to expose as a public operation. (Optional)	Optionally specify that the operation takes only input parameters but does not return any value by using the standard `@Oneway` annotation. See Section 4.3.5, "Specifying That a JWS Method Be Exposed as a Public Operation (@WebMethod and @OneWay Annotations)."
7	Add `@WebParam` annotation to customize the name of the input parameters of the exposed operations. (Optional)	See Section 4.3.6, "Customizing the Mapping Between Operation Parameters and WSDL Elements (@WebParam Annotation)."
8	Add `@WebResult` annotations to customize the name and behavior of the return value of the exposed operations. (Optional)	See Section 4.3.7, "Customizing the Mapping Between the Operation Return Value and a WSDL Element (@WebResult Annotation)."
9	Add your business code.	Add your business code to the methods to make the WebService behave as required.

4.3.1 Example of a JWS File

The following sample JWS file shows how to implement a simple Web service.

```
package examples.webservices.simple;
// Import the standard JWS annotation interfaces
```

```
import javax.jws.WebMethod;
 import javax.jws.WebService;
 import javax.jws.soap.SOAPBinding;
// Import the WebLogic-specific JWS annotation interfaces
import weblogic.jws.WLHttpTransport;
// Standard JWS annotation that specifies that the porType name of the Web
 // Service is "SimplePortType", the service name is "SimpleService", and the
 // targetNamespace used in the generated WSDL is "http://example.org"
@WebService(name="SimplePortType", serviceName="SimpleService",
            targetNamespace="http://example.org")
// Standard JWS annotation that specifies the mapping of the service onto the
 // SOAP message protocol.  In particular, it specifies that the SOAP messages
 // are document-literal-wrapped.
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
             use=SOAPBinding.Use.LITERAL,
             parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)
// WebLogic-specific JWS annotation that specifies the context path and
 // service URI used to build the URI of the Web Service is
 // "simple/SimpleService"
@WLHttpTransport(contextPath="simple", serviceUri="SimpleService",
                 portName="SimpleServicePort")
/**
  * This JWS file forms the basis of simple Java-class implemented WebLogic
  * Web Service with a single operation: sayHello
  *
  */
public class SimpleImpl {
  // Standard JWS annotation that specifies that the method should be exposed
   // as a public operation.  Because the annotation does not include the
   // member-value "operationName", the public name of the operation is the
  // same as the method name: sayHello.
  @WebMethod()
   public String sayHello(String message) {
     System.out.println("sayHello:" + message);
     return "Here is the message: '" + message + "'";
   }
 }
```

4.3.2 Specifying that the JWS File Implements a Web Service (@WebService Annotation)

Use the standard `@WebService` annotation to specify, at the class level, that the JWS file implements a Web service, as shown in the following code excerpt:

```
@WebService(name="SimplePortType", serviceName="SimpleService",
            targetNamespace="http://example.org")
```

In the example, the name of the Web service is `SimplePortType`, which will later map to the `wsdl:portType` element in the WSDL file generated by the `jwsc` Ant task. The service name is `SimpleService`, which will map to the `wsdl:service` element in the generated WSDL file. The target namespace used in the generated WSDL is `http://example.org`.

You can also specify the following additional attributes of the `@WebService` annotation:

- `endpointInterface`—Fully qualified name of an existing service endpoint interface file. This annotation allows the separation of interface definition from the implementation. If you specify this attribute, the `jwsc` Ant task does not generate

the interface for you, but assumes you have created it and it is in your CLASSPATH.

- `portname`—Name that is used in the `wsdl:port`.

None of the attributes of the `@WebService` annotation is required. See the *Web Services Metadata for the Java Platform* (JSR 181) at http://www.jcp.org/en/jsr/detail?id=181 for the default values of each attribute.

4.3.3 Specifying the Mapping of the Web Service to the SOAP Message Protocol (@SOAPBinding Annotation)

It is assumed that you want your Web service to be available over the SOAP message protocol; for this reason, your JWS file should include the standard `@SOAPBinding` annotation, at the class level, to specify the SOAP bindings of the Web service (such as, RPC-encoded or document-literal-wrapped), as shown in the following code excerpt:

```
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
             use=SOAPBinding.Use.LITERAL,
             parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)
```

In the example, the Web service uses document-wrapped-style encodings and literal message formats, which are also the default formats if you do not specify the `@SOAPBinding` annotation.

You can also use the WebLogic-specific `@weblogic.jws.soap.SOAPBinding` annotation to specify the SOAP binding at the method level; the attributes are the same as the standard `@javax.jws.soap.SOAPBinding` annotation.

You use the `parameterStyle` attribute (in conjunction with the `style=SOAPBinding.Style.DOCUMENT` attribute) to specify whether the Web service operation parameters represent the entire SOAP message body, or whether the parameters are elements wrapped inside a top-level element with the same name as the operation.

Table 4–2 Attributes of the @SOAPBinding Annotation

Attribute	Possible Values	Default Value
`style`	`SOAPBinding.Style.RPC` `SOAPBinding.Style.DOCUMENT`	`SOAPBinding.Style.DOCUMENT`
`use`	`SOAPBinding.Use.LITERAL` `SOAPBinding.Use.ENCODED`	`SOAPBinding.Use.LITERAL`
`parameterStyle`	`SOAPBinding.ParameterStyle.BARE` `SOAPBinding.ParameterStyle.WRAPPED`	`SOAPBinding.ParameterStyle.WRAPPED`

4.3.4 Specifying the Context Path and Service URI of the Web Service (@WLHttpTransport Annotation)

Use the WebLogic-specific `@WLHttpTransport` annotation to specify the context path and service URI sections of the URL used to invoke the Web service over the HTTP transport, as well as the name of the port in the generated WSDL, as shown in the following code excerpt:

```
@WLHttpTransport(contextPath="simple", serviceUri="SimpleService",
                 portName="SimpleServicePort")
```

In the example, the name of the port in the WSDL (in particular, the `name` attribute of the `<port>` element) file generated by the `jwsc` Ant task is `SimpleServicePort`. The URL used to invoke the Web service over HTTP includes a context path of `simple` and a service URI of `SimpleService`, as shown in the following example:

```
http://host:port/simple/SimpleService
```

For reference documentation on this and other WebLogic-specific annotations, see "JWS Annotation Reference" in the *WebLogic Web Services Reference*.

4.3.5 Specifying That a JWS Method Be Exposed as a Public Operation (@WebMethod and @OneWay Annotations)

Use the standard `@WebMethod` annotation to specify that a method of the JWS file should be exposed as a public operation of the Web service, as shown in the following code excerpt:

```
public class SimpleImpl {
  @WebMethod(operationName="sayHelloOperation")
   public String sayHello(String message) {
     System.out.println("sayHello:" + message);
     return "Here is the message: '" + message + "'";
   }
 ...
```

In the example, the `sayHello()` method of the `SimpleImpl` JWS file is exposed as a public operation of the Web service. The `operationName` attribute specifies, however, that the public name of the operation in the WSDL file is `sayHelloOperation`. If you do not specify the `operationName` attribute, the public name of the operation is the name of the method itself.

You can also use the `action` attribute to specify the action of the operation. When using SOAP as a binding, the value of the `action` attribute determines the value of the `SOAPAction` header in the SOAP messages.

You can specify that an operation not return a value to the calling application by using the standard `@Oneway` annotation, as shown in the following example:

```
public class OneWayImpl {
 @WebMethod()
  @Oneway()
 public void ping() {
    System.out.println("ping operation");
  }
...
```

If you specify that an operation is one-way, the implementing method is required to return `void`, cannot use a Holder class as a parameter, and cannot throw any checked exceptions.

None of the attributes of the `@WebMethod` annotation is required. See the *Web Services Metadata for the Java Platform* (JSR 181) at `http://www.jcp.org/en/jsr/detail?id=181` for the default values of each attribute, as well as additional information about the `@WebMethod` and `@Oneway` annotations.

If none of the public methods in your JWS file are annotated with the `@WebMethod` annotation, then by default *all* public methods are exposed as Web service operations.

4.3.6 Customizing the Mapping Between Operation Parameters and WSDL Elements (@WebParam Annotation)

Use the standard `@WebParam` annotation to customize the mapping between operation input parameters of the Web service and elements of the generated WSDL file, as well as specify the behavior of the parameter, as shown in the following code excerpt:

```
public class SimpleImpl {
  @WebMethod()
   @WebResult(name="IntegerOutput",
              targetNamespace="http://example.org/docLiteralBare")
   public int echoInt(
       @WebParam(name="IntegerInput",
                 targetNamespace="http://example.org/docLiteralBare")
       int input)
  {
       System.out.println("echoInt '" + input + "' to you too!");
       return input;
   }
 ...
```

In the example, the name of the parameter of the `echoInt` operation in the generated WSDL is `IntegerInput`; if the `@WebParam` annotation were not present in the JWS file, the name of the parameter in the generated WSDL file would be the same as the name of the method's parameter: `input`. The `targetNamespace` attribute specifies that the XML namespace for the parameter is `http://example.org/docLiteralBare`; this attribute is relevant only when using document-style SOAP bindings where the parameter maps to an XML element.

You can also specify the following additional attributes of the `@WebParam` annotation:

- `mode`—The direction in which the parameter is flowing (`WebParam.Mode.IN`, `WebParam.Mode.OUT`, or `WebParam.Mode.INOUT`). The OUT and INOUT modes may be specified only for parameter types that conform to the JAX-RPC definition of `Holder` types. OUT and INOUT modes are only supported for RPC-style operations or for parameters that map to headers.
- `header`—Boolean attribute that, when set to `true`, specifies that the value of the parameter should be retrieved from the SOAP header, rather than the default body.

None of the attributes of the `@WebParam` annotation is required. See the *Web Services Metadata for the Java Platform* (JSR 181) at `http://www.jcp.org/en/jsr/detail?id=181` for the default value of each attribute.

4.3.7 Customizing the Mapping Between the Operation Return Value and a WSDL Element (@WebResult Annotation)

Use the standard `@WebResult` annotation to customize the mapping between the Web service operation return value and the corresponding element of the generated WSDL file, as shown in the following code excerpt:

```
public class Simple {
  @WebMethod()
   @WebResult(name="IntegerOutput",
              targetNamespace="http://example.org/docLiteralBare")
   public int echoInt(
       @WebParam(name="IntegerInput",
```

```
                targetNamespace="http://example.org/docLiteralBare")
        int input)
  {
        System.out.println("echoInt '" + input + "' to you too!");
        return input;
   }
 ...
```

In the example, the name of the return value of the `echoInt` operation in the generated WSDL is `IntegerOutput`; if the `@WebResult` annotation were not present in the JWS file, the name of the return value in the generated WSDL file would be the hard-coded name `return`. The `targetNamespace` attribute specifies that the XML namespace for the return value is `http://example.org/docLiteralBare`; this attribute is relevant only when using document-style SOAP bindings where the return value maps to an XML element.

None of the attributes of the `@WebResult` annotation is required. See the *Web Services Metadata for the Java Platform* (JSR 181) at `http://www.jcp.org/en/jsr/detail?id=181` for the default value of each attribute.

4.4 Accessing Run-Time Information About a Web Service

The following sections describe how to access run-time information about a Web service:

- Section 4.4.1, "Using JwsContext to Access Run-Time Information"—Use the Web service context to access and change run-time information about the service in your JWS file.
- Section 4.4.2, "Using the Stub Interface to Access Run-Time Information"—Get and set properties on the Stub interface in the client file.

4.4.1 Using JwsContext to Access Run-Time Information

When a client application invokes a WebLogic Web service that was implemented with a JWS file, WebLogic Server automatically creates a *context* that the Web service can use to access, and sometimes change, run-time information about the service. Much of this information is related to conversations, such as whether the current conversation is finished, the current values of the conversational properties, changing conversational properties at run time, and so on. (See "Creating Conversational Web Services" in *Programming Advanced Features of JAX-RPC Web Services for Oracle WebLogic Server* for information about conversations and how to implement them.) Some of the information accessible via the context is more generic, such as the protocol that was used to invoke the Web service (HTTP/S or JMS), the SOAP headers that were in the SOAP message request, and so on.

You can use annotations and WebLogic Web service APIs in your JWS file to access run-time context information, as described in the following sections.

4.4.1.1 Guidelines for Accessing the Web Service Context

The following example shows a simple JWS file that uses the context to determine the protocol that was used to invoke the Web service. The code in **bold** is discussed in the programming guidelines described following the example.

```
package examples.webservices.jws_context;
import javax.jws.WebMethod;
 import javax.jws.WebService;
```

```
import weblogic.jws.WLHttpTransport;
import weblogic.jws.Context;
import weblogic.wsee.jws.JwsContext;
 import weblogic.wsee.jws.Protocol;
@WebService(name="JwsContextPortType", serviceName="JwsContextService",
            targetNamespace="http://example.org")
@WLHttpTransport(contextPath="contexts", serviceUri="JwsContext",
                 portName="JwsContextPort")
/**
  * Simple web service to show how to use the @Context annotation.
  */
public class JwsContextImpl {
  @Context
   private JwsContext ctx;
  @WebMethod()
   public String getProtocol() {
    Protocol protocol = ctx.getProtocol();
    System.out.println("protocol: " + protocol);
     return "This is the protocol: " + protocol;
   }
}
```

Use the following guidelines in your JWS file to access the run-time context of the Web service, as shown in the code in **bold** in the preceding example:

- Import the `@weblogic.jws.Context` JWS annotation:

  ```
  import weblogic.jws.Context;
  ```

- Import the `weblogic.wsee.jws.JwsContext` API, as well as any other related APIs that you might use (the example also uses the `weblogic.wsee.jws.Protocol` API):

  ```
  import weblogic.wsee.jws.JwsContext;
   import weblogic.wsee.jws.Protocol;
  ```

  See the `weblogic.wsee.*` packages in the *Java API Reference for Oracle WebLogic Server* for more documentation about the context-related APIs.

- Annotate a private variable, of data type `weblogic.wsee.jws.JwsContext`, with the field-level `@Context` JWS annotation:

  ```
  @Context
   private JwsContext ctx;
  ```

  WebLogic Server automatically assigns the annotated variable (in this case, `ctx`) with a run-time implementation of `JwsContext` the first time the Web service is invoked, which is how you can later use the variable without explicitly initializing it in your code.

  Use the methods of the `JwsContext` class to access run-time information about the Web service. The following example shows how to get the protocol that was used to invoke the Web service.

  ```
  Protocol protocol = ctx.getProtocol();
  ```

  See Section 4.4.1.2, "Methods of the JwsContext" for the full list of available methods.

4.4.1.2 Methods of the JwsContext

The following table summarizes the methods of the `JwsContext` that you can use in your JWS file to access run-time information about the Web service. See

`weblogic.wsee.*` packages in the *Java API Reference for Oracle WebLogic Server* for detailed reference information about `JwsContext`, and other context-related APIs, as `Protocol` and `ServiceHandle`.

Table 4–3 Methods of JwsContext

Method	Returns	Description
`isFinished()`	`boolean`	Returns a boolean value specifying whether the current conversation is finished, or if it is still continuing. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`finishConversation()`	`void`	Finishes the current conversation. This method is equivalent to a client application invoking a method that has been annotated with the `@Conversation (Conversation.Phase.FINISH)` JWS annotation. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`setMaxAge(java.util.Date)`	`void`	Sets a new maximum age for the conversation to an absolute `Date`. If the date parameter is in the past, WebLogic Server immediately finishes the conversation. This method is equivalent to the `maxAge` attribute of the `@Conversational` annotation, which specifies the *default* maximum age of a conversation. Use this method to override this default value at run time. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`setMaxAge(String)`	`void`	Sets a new maximum age for the conversation by specifying a `String` duration, such as `1 day`. Valid values for the `String` parameter are a number and one of the following terms: ■ `seconds` ■ `minutes` ■ `hours` ■ `days` ■ `years` For example, to specify a maximum age of ten minutes, use the following syntax: `ctx.setMaxAge("10 minutes")` This method is equivalent to the `maxAge` attribute of the `@Conversational` annotation, which specifies the *default* maximum age of a conversation. Use this method to override this default value at run time. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.

Table 4–3 (Cont.) Methods of JwsContext

Method	Returns	Description
`getMaxAge()`	`long`	Returns the maximum allowed age, in seconds, of a conversation. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`getCurrentAge()`	`long`	Returns the current age, in seconds, of the conversation. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`resetIdleTime()`	`void`	Resets the timer which measures the number of seconds since the last activity for the current conversation. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`setMaxIdleTime(long)`	`void`	Sets the number of seconds that the conversation can remain idle before WebLogic Server finishes it due to client inactivity. This method is equivalent to the `maxIdleTime` attribute of the `@Conversational` annotation, which specifies the *default* idle time of a conversation. Use this method to override this default value at run time. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`setMaxIdleTime(String)`	`void`	Sets the number of seconds, specified as a `String`, that the conversation can remain idle before WebLogic Server finishes it due to client inactivity. Valid values for the `String` parameter are a number and one of the following terms: ■ `seconds` ■ `minutes` ■ `hours` ■ `days` ■ `years` For example, to specify a maximum idle time of ten minutes, use the following syntax: `ctx.setMaxIdleTime("10 minutes")` This method is equivalent to the `maxIdleTime` attribute of the `@Conversational` annotation, which specifies the *default* idle time of a conversation. Use this method to override this default value at run time. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.

Table 4–3 (Cont.) Methods of JwsContext

Method	Returns	Description
`getMaxIdleTime()`	`long`	Returns the number of seconds that the conversation is allowed to remain idle before WebLogic Server finishes it due to client inactivity. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`getCurrentIdleTime()`	`long`	Gets the number of seconds since the last client request, or since the conversation's maximum idle time was reset. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`getCallerPrincipal()`	`java.security.Principal`	Returns the security principal associated with the operation that was just invoked, assuming that basic authentication was performed.
`isCallerInRole(String)`	`boolean`	Returns `true` if the authenticated principal is within the specified security role.
`getService()`	`weblogic.wsee.jws.ServiceHandle`	Returns an instance of `ServiceHandle`, a WebLogic Web service API, which you can query to gather additional information about the Web service, such as the conversation ID (if the Web service is conversational), the URL of the Web service, and so on.
`getLogger(String)`	`weblogic.wsee.jws.util.Logger`	Gets an instance of the `Logger` class, which you can use to send messages from the Web service to a log file.
`getInputHeaders()`	`org.w3c.dom.Element[]`	Returns an array of the SOAP headers associated with the SOAP request message of the current operation invoke.
`setUnderstoodInputHeaders(boolean)`	`void`	Indicates whether input headers should be understood.
`getUnderstoodInputHeaders()`	`boolean`	Returns the value that was most recently set by a call to setUnderstoodInputHeader.
`setOutputHeaders(Element[])`	`void`	Specifies an array of SOAP headers that should be associated with the outgoing SOAP response message sent back to the client application that initially invoked the current operation.
`getProtocol()`	`weblogic.wsee.jws.Protocol`	Returns the protocol (such as HTTP/S or JMS) used to invoke the current operation.

4.4.2 Using the Stub Interface to Access Run-Time Information

The`javax.xml.rpc.Stub` interface enables you to dynamically configure the Stub instance in your Web service client file. For more information, see http://download.oracle.com/javaee/6/api/javax/xml/rpc/Stub.html. For example, you can set the target service endpoint dynamically for the `port` Stub instance, as follows:

```
ComplexService service = new ComplexService_Impl (args[0] + "?WSDL" );
 ComplexPortType port = service.getComplexServicePort();
 ((Stub)port)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY,
    "http://localhost:8010/MyContext/MyService");
```

For more information about developing Web service clients, see Chapter 6, "Developing JAX-RPC Web Service Clients."

The following table summarizes the methods of the `Stub` interface that you can use in your JWS file to access run-time information about the Web service.

Table 4–4 Methods of Stub Interface

Method	Returns	Description
`_getProperty()`	`java.lang.Object`	Gets the value of the specified configuration property.
`_getPropertyNames()`	`java.util.Iterator`	Returns an Iterator view of the names of the properties that can be configured on the `Stub` instance.
`_setProperty()`	`void`	Sets the name and value of a configuration property for the `Stub` instance.

The following table defined the `javax.xml.rpc.Stub` property values that you can access from the Stub instance.

Table 4–5 Properties of Stub Interface

Property	Type	Description
`ENDPOINT_ADDRESS_PROPERTY`	`java.lang.String`	Target service endpoint address.
`PASSWORD_PROPERTY`	`java.lang.String`	Password used for authentication.
`SESSION_MAINTAIN_PROPERTY`	`java.lang.String`	Flag specifying whether to participate in a session with a service endpoint.
`USERNAME_PROPERTY`	`java.lang.String`	User name used for authentication.

4.5 Should You Implement a Stateless Session EJB?

The `jwsc` Ant task always chooses a plain Java object as the underlying implementation of a Web service when processing your JWS file.

Sometimes, however, you might want the underlying implementation of your Web service to be a stateless session EJB so as to take advantage of all that EJBs have to offer, such as instance pooling, transactions, security, container-managed persistence, container-managed relationships, and data caching. If you decide you want an EJB implementation for your Web service, then follow the programming guidelines in the following section.

Note: JAX-RPC supports EJB 2.x only; it does not support EJB 3.0.

4.5.1 Programming Guidelines When Implementing an EJB in Your JWS File

The general guideline is to always use EJBGen annotations in your JWS file to automatically generate, rather than manually create, the EJB Remote and Home interface classes and deployment descriptor files needed when implementing an EJB. EJBGen annotations work in the same way as JWS annotations: they follow the JDK 5.0 metadata syntax and greatly simplify your programming tasks.

For more information on EJBGen, see "EJBGen Reference" in *Developing Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server*.

Follow these guidelines when explicitly implementing a stateless session EJB in your JWS file. See Section 4.5.2, "Example of a JWS File That Implements an EJB" for an example; the relevant sections are shown in **bold**:

- Import the standard Java Platform, Enterprise Edition (Java EE) Version 5 EJB classes:

```
import javax.ejb.SessionBean;
 import javax.ejb.SessionContext;
```

- Import the EJBGen annotations, all of which are in the `weblogic.ejbgen` package. At a minimum you need to import the `@Session` annotation; if you want to use additional EJBGen annotations in your JWS file to specify the shape and behavior of the EJB, see the "EJBGen Reference" in *Developing Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server* for the name of the annotation you should import.

```
import weblogic.ejbgen.Session;
```

- At a minimum, use the `@Session` annotation at the class level to specify the name of the EJB:

```
@Session(ejbName="TransactionEJB")
```

  `@Session` is the only required EJBGen annotation when used in a JWS file. You can, if you want, use other EJBGen annotations to specify additional features of the EJB.

- Ensure that the JWS class implements `SessionBean`:

```
public class TransactionImpl implements SessionBean {...
```

- You must also include the standard EJB methods `ejbCreate()`, `ejbActivate()` and so on, although you typically do not need to add code to these methods unless you want to change the default behavior of the EJB:

```
  public void ejbCreate() {}
   public void ejbActivate() {}
   public void ejbRemove() {}
   public void ejbPassivate() {}
   public void setSessionContext(SessionContext sc) {}
```

If you follow all these guidelines in your JWS file, the `jwsc` Ant task later compiles the Web service into an EJB and packages it into an EJB JAR file inside of the Enterprise Application.

4.5.2 Example of a JWS File That Implements an EJB

The following example shows a simple JWS file that implement a stateless session EJB. The relevant code is shown in **bold**.

```
package examples.webservices.transactional;
import javax.ejb.SessionBean;
 import javax.ejb.SessionContext;
import javax.jws.WebMethod;
 import javax.jws.WebService;
import weblogic.jws.WLHttpTransport;
 import weblogic.jws.Transactional;
import weblogic.ejbgen.Session;
@Session(ejbName="TransactionEJB")
@WebService(name="TransactionPortType", serviceName="TransactionService",
```

```
            targetNamespace="http://example.org")
@WLHttpTransport(contextPath="transactions", serviceUri="TransactionService",
                 portName="TransactionPort")
/**
  * This JWS file forms the basis of simple EJB-implemented WebLogic
  * Web Service with a single operation: sayHello.  The operation executes
  * as part of a transaction.
  *
  */
public class TransactionImpl implements SessionBean {
  @WebMethod()
   @Transactional(value=true)
  public String sayHello(String message) {
     System.out.println("sayHello:" + message);
     return "Here is the message: '" + message + "'";
   }
  // Standard EJB methods.  Typically there's no need to override the methods.
  public void ejbCreate() {}
   public void ejbActivate() {}
   public void ejbRemove() {}
   public void ejbPassivate() {}
   public void setSessionContext(SessionContext sc) {}
 }
```

4.6 Programming the User-Defined Java Data Type

The methods of the JWS file that are exposed as Web service operations do not necessarily take built-in data types (such as Strings and integers) as parameters and return values, but rather, might use a Java data type that you create yourself. An example of a user-defined data type is `TradeResult`, which has two fields: a `String` stock symbol and an integer number of shares traded.

If your JWS file uses user-defined data types as parameters or return values of one or more of its methods, you must create the Java code of the data type yourself, and then import the class into your JWS file and use it appropriately. The `jwsc` Ant task will later take care of creating all the necessary data binding artifacts, such as the corresponding XML Schema representation of the Java user-defined data type, the JAX-RPC type mapping file, and so on.

Follow these basic requirements when writing the Java class for your user-defined data type:

- Define a default constructor, which is a constructor that takes no parameters.
- Define both `getXXX()` and `setXXX()` methods for each member variable that you want to publicly expose.
- Make the data type of each exposed member variable one of the built-in data types, or another user-defined data type that consists of built-in data types.

These requirements are specified by JAX-RPC; for more detailed information and the complete list of requirements, see the JAX-RPC specification at `http://java.net/projects/jax-rpc/`.

The `jwsc` Ant task can generate data binding artifacts for most common XML and Java data types. For the list of supported user-defined data types, see Section 5.3, "Supported User-Defined Data Types." See Section 5.2, "Supported Built-In Data Types" for the full list of supported built-in data types.

The following example shows a simple Java user-defined data type called `BasicStruct`:

```
package examples.webservices.complex;
/**
  * Defines a simple JavaBean called BasicStruct that has integer, String,
  * and String[] properties
  */
public class BasicStruct {
  // Properties
  private int intValue;
   private String stringValue;
   private String[] stringArray;
  // Getter and setter methods
  public int getIntValue() {
     return intValue;
   }
  public void setIntValue(int intValue) {
     this.intValue = intValue;
   }
  public String getStringValue() {
     return stringValue;
   }
  public void setStringValue(String stringValue) {
     this.stringValue = stringValue;
   }
  public String[] getStringArray() {
     return stringArray;
   }
  public void setStringArray(String[] stringArray) {
     this.stringArray = stringArray;
   }
}
```

The following snippets from a JWS file show how to import the `BasicStruct` class and use it as both a parameter and return value for one of its methods; for the full JWS file, see Section 2.2.2, "Sample ComplexImpl.java JWS File":

```
package examples.webservices.complex;
// Import the standard JWS annotation interfaces
import javax.jws.WebMethod;
 import javax.jws.WebParam;
 import javax.jws.WebResult;
 import javax.jws.WebService;
 import javax.jws.soap.SOAPBinding;
// Import the WebLogic-specific JWS annotation interface
import weblogic.jws.WLHttpTransport;
// Import the BasicStruct JavaBean
import examples.webservices.complex.BasicStruct;
@WebService(serviceName="ComplexService", name="ComplexPortType",
             targetNamespace="http://example.org")
...
public class ComplexImpl {
  @WebMethod(operationName="echoComplexType")
  public BasicStruct echoStruct(BasicStruct struct)
  {
     return struct;
   }
 }
```

4.7 Throwing Exceptions

When you write the error-handling Java code in methods of the JWS file, you can either throw your own user-defined exceptions or throw a `javax.xml.rpc.soap.SOAPFaultException` exception. If you throw a `SOAPFaultException`, WebLogic Server maps it to a SOAP fault and sends it to the client application that invokes the operation.

If your JWS file throws any type of Java exception other than `SOAPFaultException`, WebLogic Server tries to map it to a SOAP fault as best it can. However, if you want to control what the client application receives and send it the best possible exception information, you should explicitly throw a `SOAPFaultException` exception or one that extends the exception. See the JAX-RPC specification at http://java.net/projects/jax-rpc/ for detailed information about creating and throwing your own user-defined exceptions.

The following excerpt describes the `SOAPFaultException` class:

```
public class SOAPFaultException extends java.lang.RuntimeException {
     public SOAPFaultException (QName faultcode,
                                String faultstring,
                                String faultactor,
                                javax.xml.soap.Detail detail ) {...}
     public Qname getFaultCode() {...}
     public String getFaultString() {...}
     public String getFaultActor() {...}
     public javax.xml.soap.Detail getDetail() {...}
 }
```

Use the SOAP with Attachments API for Java 1.1 (SAAJ) `javax.xml.soap.SOAPFactory.createDetail()` method to create the `Detail` object, which is a container for `DetailEntry` objects that provide detailed application-specific information about the error.

You can use your own implementation of the `SOAPFactory`, or use Oracle 's, which can be accessed in the JWS file by calling the static method `weblogic.wsee.util.WLSOAPFactory.createSOAPFactory()` which returns a `javax.xml.soap.SOAPFactory object`. Then at run time, use the `-Djavax.xml.soap.SOAPFactory` flag to specify Oracle's `SOAPFactory` implementation as shown:

```
-Djavax.xml.soap.SOAPFactory=weblogic.xml.saaj.SOAPFactoryImpl
```

The following JWS file shows an example of creating and throwing a `SOAPFaultException` from within a method that implements an operation of your Web service; the sections in bold highlight the exception code:

```
package examples.webservices.soap_exceptions;
import javax.xml.namespace.QName;
 import javax.xml.soap.Detail;
 import javax.xml.soap.SOAPException;
 import javax.xml.soap.SOAPFactory;
 import javax.xml.rpc.soap.SOAPFaultException;
// Import the @WebService annotation
import javax.jws.WebService;
// Import WLHttpTransport
import weblogic.jws.WLHttpTransport;
@WebService(serviceName="SoapExceptionsService",
             name="SoapExceptionsPortType",
             targetNamespace="http://example.org")
@WLHttpTransport(contextPath="exceptions",
```

```
                serviceUri="SoapExceptionsService",
                portName="SoapExceptionsServicePort")
 /**
  * This JWS file forms the basis of simple Java-class implemented WebLogic
  * Web Service with a single operation: sayHelloWorld
  *
  */
public class SoapExceptionsImpl {
  public SoapExceptionsImpl() {
  }
  public void tirarSOAPException() {
    Detail detail = null;
    try {
      SOAPFactory soapFactory = SOAPFactory.newInstance();
       detail = soapFactory.createDetail();
    } catch (SOAPException e) {
         // do something
     }
    QName faultCode = null;
     String faultString = "the fault string";
     String faultActor = "the fault actor";
     throw new SOAPFaultException(faultCode, faultString, faultActor, detail);
   }
 }
```

The preceding example uses the default implementation of `SOAPFactory`.

> **Note:** If you create and throw your own exception (rather than use `SOAPFaultException`) and two or more of the properties of your exception class are of the same data type, then you *must* also create setter methods for these properties, even though the JAX-RPC specification does not require it. This is because when a WebLogic Web service receives the exception in a SOAP message and converts the XML into the Java exception class, there is no way of knowing which XML element maps to which class property without the corresponding setter methods.

4.8 Invoking Another Web Service from the JWS File

From within your JWS file you can invoke another Web service, either one deployed on WebLogic Server or one deployed on some other application server, such as .NET. The steps to do this are similar to those described in Section 2.4, "Invoking a Web Service from a Java SE Client," except that rather than running the `clientgen` Ant task to generate the client stubs, you include a `<clientgen>` child element of the `jwsc` Ant task that builds the invoking Web service to generate the client stubs instead. You then use the standard JAX-RPC APIs in your JWS file.

See Section 6.3, "Invoking a Web Service from Another Web Service" for detailed instructions.

4.9 Programming Additional Miscellaneous Features Using JWS Annotations and APIs

The following sections describe additional miscellaneous features you can program by specifying particular JWS annotations in your JWS file or using WebLogic Web services APIs:

4.9.1 Sending Binary Data Using MTOM/XOP

SOAP Message Transmission Optimization Mechanism/XML-binary Optimized Packaging (MTOM/XOP) describes a method for optimizing the transmission of XML data of type `xs:base64Binary` in SOAP messages. When the transport protocol is HTTP, MIME attachments are used to carry that data while at the same time allowing both the sender and the receiver direct access to the XML data in the SOAP message without having to be aware that any MIME artifacts were used to marshal the `xs:base64Binary` data. The binary data optimization process involves encoding the binary data, removing it from the SOAP envelope, compressing it and attaching it to the MIME package, and adding references to that package in the SOAP envelope.

The MTOM specification does not require that, when MTOM is enabled, the Web service run time use XOP binary optimization when transmitting `base64binary` data. Rather, the specification allows the run time to choose to do so. This is because in certain cases the run time may decide that it is more efficient to send `base64binary` data directly in the SOAP Message; an example of such a case is when transporting small amounts of data in which the overhead of conversion and transport consumes more resources than just inlining the data as is. The WebLogic Web services implementation for MTOM for JAX-RPC service, however, *always* uses MTOM/XOP when MTOM is enabled.

Support for MTOM/XOP in WebLogic JAX-RPC Web services is implemented using the pre-packaged WS-Policy file `Mtom.xml`. WS-Policy files follow the *WS-Policy* specification, described at `http://www.w3.org/TR/ws-policy`; this specification provides a general purpose model and XML syntax to describe and communicate the policies of a Web service, in this case the use of MTOM/XOP to send binary data. The installation of the pre-packaged `Mtom.xml` WS-Policy file in the `types` section of the Web service WSDL is as follows (provided for your information only; you cannot change this file):

```
<wsp:Policy wsu:Id="myService_policy">
     <wsp:ExactlyOne>
         <wsp:All>
             <wsoma:OptimizedMimeSerialization

xmlns:wsoma="http://schemas.xmlsoap.org/ws/2004/09/policy/optimizedmimeserializati
on" />
         </wsp:All>
     </wsp:ExactlyOne>
 </wsp:Policy>
```

When you deploy the compiled JWS file to WebLogic Server, the dynamic WSDL will automatically contain the following snippet that references the MTOM WS-Policy file; the snippet indicates that the Web service uses MTOM/XOP:

```
<wsdl:binding name="BasicHttpBinding_IMtomTest"
                        type="i0:IMtomTest">
     <wsp:PolicyReference URI="#myService_policy" />
     <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
```

You can associate the `Mtom.xml` WS-Policy file with a Web service at development-time by specifying the `@Policy` metadata annotation in your JWS file. Be sure you also specify the `attachToWsdl=true` attribute to ensure that the dynamic WSDL includes the required reference to the `Mtom.xml` file; see the example below.

You can associate the `Mtom.xml` WS-Policy file with a Web service at deployment time by modifying the WSDL to add the Policy to the types section just before deployment.

In addition, you can attach the file at run time using by the Administration Console; for details, see "Attach a WS-Policy file to a Web service" in the *Oracle WebLogic Server Administration Console Online Help*. This section describes how to use the JWS annotation.

Note: In this release of WebLogic Server, the only supported Java data type when using MTOM/XOP is `byte[]`; other binary data types, such as `image`, are not supported.

In addition, this release of WebLogic Server does not support using MTOM with deprecated 9.*x* security policies.

To send binary data using MTOM/XOP, follow these steps:

1. Use the WebLogic-specific `@weblogic.jws.Policy` annotation in your JWS file to specify that the pre-packaged `Mtom.xml` file should be applied to your Web service, as shown in the following simple JWS file (relevant code shown in bold):

```
package examples.webservices.mtom;
import javax.jws.WebMethod;
 import javax.jws.WebService;
import weblogic.jws.WLHttpTransport;
 import weblogic.jws.Policy;
@WebService(name="MtomPortType",
             serviceName="MtomService",
             targetNamespace="http://example.org")
@WLHttpTransport(contextPath="mtom",
                  serviceUri="MtomService",
                  portName="MtomServicePort")
@Policy(uri="policy:Mtom.xml", attachToWsdl=true)
public class MtomImpl {
  @WebMethod
   public String echoBinaryAsString(byte[] bytes) {
     return new String(bytes);
   }
```

2. Use the Java `byte[]` data type in your Web service operations as either a return value or input parameter whenever you want the resulting SOAP message to use MTOM/XOP to send or receive the binary data. See the implementation of the `echoBinaryAsString` operation above for an example; this operation simply takes as input an array of `byte` and returns it as a `String`.
3. The WebLogic Web services run time has built in MTOM/XOP support which is enabled if the WSDL for which the `clientgen` Ant task generates client-side artifacts specifies MTOM/XOP support. In your client application itself, simply invoke the operations as usual, using `byte[]` as the relevant data type.

See the *SOAP Message Transmission Optimization Mechanism* specification at `http://www.w3.org/TR/2005/REC-soap12-mtom-20050125` for additional

information about the MTOM/XOP feature itself as well as the version of the specification supported by WebLogic JAX-RPC Web services.

4.9.2 Streaming SOAP Attachments

Using the `@weblogic.jws.StreamAttachments` JWS annotation, you can specify that a Web service use a streaming API when reading inbound SOAP messages that include attachments, rather than the default behavior in which the service reads the entire message into memory. This feature increases the performance of Web services whose SOAP messages are particular large.

See "weblogic.jws.StreamAttachments" in the *WebLogic Web Services Reference for Oracle WebLogic Server* for an example of specifying that attachments should be streamed.

4.9.3 Using SOAP 1.2

WebLogic Web services use, by default, Version 1.1 of Simple Object Access Protocol (SOAP) as the message format when transmitting data and invocation calls between the Web service and its client. WebLogic Web services support both SOAP 1.1 and the newer SOAP 1.2, and you are free to use either version.

To specify that the Web service use Version 1.2 of SOAP, use the class-level `@weblogic.jws.Binding` annotation in your JWS file and set its single attribute to the value `Binding.Type.SOAP12`, as shown in the following example (relevant code shown in **bold**):

```
package examples.webservices.soap12;
import javax.jws.WebMethod;
 import javax.jws.WebService;
import weblogic.jws.WLHttpTransport;
 import weblogic.jws.Binding;
@WebService(name="SOAP12PortType",
             serviceName="SOAP12Service",
             targetNamespace="http://example.org")
@WLHttpTransport(contextPath="soap12",
                  serviceUri="SOAP12Service",
                  portName="SOAP12ServicePort")
@Binding(Binding.Type.SOAP12)
/**
  * This JWS file forms the basis of simple Java-class implemented WebLogic
  * Web Service with a single operation: sayHello.  The class uses SOAP 1.2
  * as its binding.
  *
  */
public class SOAP12Impl {
  @WebMethod()
   public String sayHello(String message) {
     System.out.println("sayHello:" + message);
     return "Here is the message: '" + message + "'";
   }
 }
```

Other than set this annotation, you do not have to do anything else for the Web service to use SOAP 1.2, including changing client applications that invoke the Web service; the WebLogic Web services run time takes care of all the rest.

See "weblogic.jws.Binding" in the *WebLogic Web Services Reference for Oracle WebLogic Server* for additional information about this annotation.

4.9.4 Specifying that Operations Run Inside of a Transaction

When a client application invokes a WebLogic Web service operation, the operation invocation takes place outside the context of a transaction, by default. If you want the operation to run inside a transaction, specify the `@weblogic.jws.Transactional` annotation in your JWS file, and set the boolean `value` attribute to `true`, as shown in the following example (relevant code shown in **bold**):

```
package examples.webservices.transactional;
import javax.jws.WebMethod;
 import javax.jws.WebService;
import weblogic.jws.WLHttpTransport;
 import weblogic.jws.Transactional;
@WebService(name="TransactionPojoPortType",
             serviceName="TransactionPojoService",
             targetNamespace="http://example.org")
@WLHttpTransport(contextPath="transactionsPojo",
                  serviceUri="TransactionPojoService",
                  portName="TransactionPojoPort")
/**
  * This JWS file forms the basis of simple WebLogic
  * Web Service with a single operation: sayHello.  The operation executes
  * as part of a transaction.
  *
  */
public class TransactionPojoImpl {
  @WebMethod()
   @Transactional(value=true)
   public String sayHello(String message) {
     System.out.println("sayHello:" + message);
     return "Here is the message: '" + message + "'";
   }
 }
```

If you want *all* operations of a Web service to run inside of a transaction, specify the `@Transactional` annotation at the class-level. If you want only a subset of the operations to be transactional, specify the annotation at the method-level. If there is a conflict, the method-level value overrides the class-level.

See "weblogic.jws.Transactional" in the *WebLogic Web Services Reference for Oracle WebLogic Server* for information about additional attributes.

4.9.5 Getting the HttpServletRequest/Response Object

If your Web service uses HTTP as its transport protocol, you can use the `"weblogic.wsee.connection.transport.servlet.HttpTransportUtils"` API in the *Java API Reference for Oracle WebLogic Server*to get the `javax.servlet.http.HttpServletRequest` and `javax.servlet.http.HttpServletResponse` objects from the JAX-RPC `ServletEndpointContext` object, as shown in the following example (relevant code shown in bold and explained after the example):

```
package examples.webservices.http_transport_utils;
import javax.xml.rpc.server.ServiceLifecycle;
 import javax.xml.rpc.server.ServletEndpointContext;
 import javax.xml.rpc.ServiceException;
import javax.servlet.http.HttpServletRequest;
 import javax.servlet.http.HttpServletResponse;
import javax.jws.WebMethod;
 import javax.jws.WebService;
```

```
import weblogic.jws.WLHttpTransport;
import weblogic.wsee.connection.transport.servlet.HttpTransportUtils;
@WebService(name="HttpTransportUtilsPortType",
            serviceName="HttpTransportUtilsService",
            targetNamespace="http://example.org")
@WLHttpTransport(contextPath="servlet", serviceUri="HttpTransportUtils",
                 portName="HttpTransportUtilsPort")
public class HttpTransportUtilsImpl implements ServiceLifecycle {
  private ServletEndpointContext wsctx = null;
  public void init(Object context) throws ServiceException {
     System.out.println("ServletEndpointContext inited...");
     wsctx = (ServletEndpointContext)context;
   }
  public void destroy() {
     System.out.println("ServletEndpointContext destroyed...");
     wsctx = null;
   }
  @WebMethod()
   public String getServletRequestAndResponse() {
    HttpServletRequest request =
        HttpTransportUtils.getHttpServletRequest(wsctx.getMessageContext());
    HttpServletResponse response =
        HttpTransportUtils.getHttpServletResponse(wsctx.getMessageContext());
    System.out.println("HttpTransportUtils API used successfully.");
     return "HttpTransportUtils API used successfully";
  }
}
```

The important parts of the preceding example are as follows:

- Import the required JAX-RPC and Servlet classes:

  ```
  import javax.xml.rpc.server.ServiceLifecycle;
   import javax.xml.rpc.server.ServletEndpointContext;
   import javax.xml.rpc.ServiceException;
  import javax.servlet.http.HttpServletRequest;
   import javax.servlet.http.HttpServletResponse;
  ```

- Import the WebLogic `HttpTransportUtils` class:

  ```
  import weblogic.wsee.connection.transport.servlet.HttpTransportUtils;
  ```

- Because you will be querying the JAX-RPC message context, your JWS file must explicitly implement `ServiceLifecycle`:

  ```
  public class HttpTransportUtilsImpl implements ServiceLifecycle
  ```

- Create a variable of data type `ServletEndpointContext`:

  ```
   private ServletEndpointContext wsctx = null;
  ```

- Because the JWS file implements `ServiceLifecycle`, you must also implement the `init` and `destroy` lifecycle methods:

  ```
    public void init(Object context) throws ServiceException {
       System.out.println("ServletEndpointContext inited...");
       wsctx = (ServletEndpointContext)context;
     }
     public void destroy() {
       System.out.println("ServletEndpointContext destroyed...");
       wsctx = null;
     }
  ```

- Finally, in the method that implements the Web service operation, use the `ServletEndpointContext` object to get the `HttpServletRequest` and `HttpServletResponse` objects:

```
HttpServletRequest request =
   HttpTransportUtils.getHttpServletRequest(wsctx.getMessageContext());
 HttpServletResponse response =
   HttpTransportUtils.getHttpServletResponse(wsctx.getMessageContext());
```

4.10 JWS Programming Best Practices

The following list provides some best practices when programming the JWS file:

- When you create a document-literal-bare Web service, use the `@WebParam` JWS annotation to ensure that all input parameters for all operations of a given Web service have a unique name. Because of the nature of document-literal-bare Web services, if you do not explicitly use the `@WebParam` annotation to specify the name of the input parameters, WebLogic Server creates one for you and run the risk of duplicating the names of the parameters across a Web service.
- In general, document-literal-wrapped Web services are the most interoperable type of Web service.
- Use the `@WebResult` JWS annotation to explicitly set the name of the returned value of an operation, rather than always relying on the hard-coded name `return`, which is the default name of the returned value if you do not use the `@WebResult` annotation in your JWS file.
- Use `SOAPFaultExceptions` in your JWS file if you want to control the exception information that is passed back to a client application when an error is encountered while invoking a the Web service.
- Even though it is not required, Oracle recommends you always specify the `portName` attribute of the WebLogic-specific `@WLHttpTransport` annotation in your JWS file. If you do not specify this attribute, the `jwsc` Ant task will generate a port name for you when generating the WSDL file, but this name might not be very user-friendly. A consequence of this is that the `getXXX()` method you use in your client applications to invoke the Web service will not be very well-named. To ensure that your client applications use the most user-friendly methods possible when invoking the Web service, specify a relevant name of the Web service port by using the `portName` attribute.

5 Understanding Data Binding

This chapter describes the data binding and the data types (both built-in and user-defined) that are supported for WebLogic Web services using Java API for XML-based RPC (JAX-RPC).

This chapter includes the following topics:

5.1 Overview of Data Binding

With the emergence of XML as the standard for exchanging data across disparate systems, Web service applications need a way to access documents that are in XML format directly from the Java application. Specifically, the XML content needs to be converted to a format that is readable by the Java application. *Data binding* describes the conversion of data between its XML and Java representations.

As in previous releases, WebLogic Web services support a full set of built-in XML Schema, Java, and SOAP types, as specified by the JAX-RPC specification, described at `http://java.net/projects/jax-rpc/`, that you can use in your Web service operations without performing any additional programming steps. Built-in data types are those such as `integer`, `string`, and `time`.

Additionally, you can use a variety of user-defined XML and Java data types, including Apache XmlBeans (in package `org.apache.xmlbeans`), as input parameters and return values of your Web service. User-defined data types are those that you create from XML Schema or Java building blocks, such as `<xsd:complexType>` or JavaBeans. The WebLogic Web services Ant tasks, such as `jwsc` and `clientgen`, automatically generate the data binding artifacts needed to convert the user-defined data types between their XML and Java representations. The XML representation is used in the SOAP request and response messages, and the Java representation is used in the JWS that implements the Web service.

Note: As of WebLogic Server 9.1, using XMLBeans 1.*x* data types (in other words, extensions of `com.bea.xml.XmlObject`) as parameters or return types of a WebLogic Web service is deprecated. New applications should use XMLBeans 2.*x* data types.

If a Web service uses XMLBeans that are compiled with the `-noupa` option, then `-Dweblogic.wsee.bind.setCompileNoUpaRule=true` flag is required to be set in the WebLogic server startup script to ensure the Web service deploys successfully.Otherwise, deployment will fail with the following error: `cos-nonambig: Content model violates the unique particle attribution rule`.

5.2 Supported Built-In Data Types

The following sections describe the built-in data types supported by WebLogic Web services and the mapping between their XML and Java representations. As long as the data types of the parameters and return values of the back-end components that implement your Web service are in the set of built-in data types, WebLogic Server automatically converts the data between XML and Java.

If, however, you use user-defined data types, then you must create the data binding artifacts that convert the data between XML and Java.WebLogic Server includes the `jwsc` and `wsdlc` Ant tasks that can automatically generate the data binding artifacts for most user-defined data types. See Section 5.3, "Supported User-Defined Data Types" for a list of supported XML and Java data types.

5.2.1 XML-to-Java Mapping for Built-in Data Types

The following table lists the supported XML Schema data types (target namespace `http://www.w3.org/2001/XMLSchema`) and their corresponding Java data types.

For a list of the supported user-defined XML data types, see Section 5.2.2, "Java-to-XML Mapping for Built-In Data Types."

Table 5–1 Mapping XML Schema Built-in Data Types to Java Data Types

XML Schema Data Type	Equivalent Java Data Type (lower case indicates a primitive data type)
`boolean`	`boolean`
`byte`	`byte`
`short`	`short`
`int`	`int`
`long`	`long`
`float`	`float`
`double`	`double`
`integer`	`java.math.BigInteger`
`decimal`	`java.math.BigDecimal`
`string`	`java.lang.String`
`dateTime`	`java.util.Calendar`

Table 5–1 (Cont.) Mapping XML Schema Built-in Data Types to Java Data Types

XML Schema Data Type	Equivalent Java Data Type (lower case indicates a primitive data type)
base64Binary	byte[]
hexBinary	byte[]
duration	java.lang.String
time	java.util.Calendar
date	java.util.Calendar
gYearMonth	java.util.Calendar
gYear	java.util.Calendar
gMonthDay	java.util.Calendar
gDay	java.util.Calendar
gMonth	java.util.Calendar
anyURI	java.net.URI
NOTATION	java.lang.String
token	java.lang.String
normalizedString	java.lang.String
language	java.lang.String
Name	java.lang.String
NMTOKEN	java.lang.String
NCName	java.lang.String
NMTOKENS	java.lang.String[]
ID	java.lang.String
IDREF	java.lang.String
ENTITY	java.lang.String
IDREFS	java.lang.String[]
ENTITIES	java.lang.String[]
nonPositiveInteger	java.math.BigInteger
nonNegativeInteger	java.math.BigInteger
negativeInteger	java.math.BigInteger
unsignedLong	java.math.BigInteger
positiveInteger	java.math.BigInteger
unsignedInt	long
unsignedShort	int
unsignedByte	short
Qname	javax.xml.namespace.QName

5.2.2 Java-to-XML Mapping for Built-In Data Types

For a list of the supported user-defined Java data types, see Section 5.3.2, "Supported Java User-Defined Data Types."

Table 5–2 Mapping Java Data Types to XML Schema Data Types

Java Data Type (lower case indicates a primitive data type)	Equivalent XML Schema Data Type
int	int
short	short
long	long
float	float
double	double
byte	byte
boolean	boolean
char	string (with facet of length=1)
java.lang.Integer	int
java.lang.Short	short
java.lang.Long	long
java.lang.Float	float
java.lang.Double	double
java.lang.Byte	byte
java.lang.Boolean	boolean
java.lang.Character	string (with facet of length=1)
java.lang.String	string
java.math.BigInteger	integer
java.math.BigDecimal	decimal
java.util.Calendar	dateTime
java.util.Date	dateTime
byte[]	base64Binary
javax.xml.namespace.QName	Qname
java.net.URI	anyURI
javax.xml.datatype.XMLGregorianCalendar	anySimpleType
javax.xml.datatype.Duration	duration
java.lang.Object	anyType
java.awt.Image	base64Binary
javax.activation.DataHandler	base64Binary
javax.xml.transform.Source	base64Binary
java.util.UUID	string

5.3 Supported User-Defined Data Types

The tables in the following sections list the user-defined XML and Java data types for which the `jwsc` and `wsdlc` Ant tasks can automatically generate data binding artifacts, such as the corresponding Java or XML representation, the JAX-RPC type mapping file, and so on.

If your XML or Java data type is not listed in these tables, and it is not one of the built-in data types listed in Section 5.2, "Supported Built-In Data Types," then you must create the user-defined data type artifacts manually.

5.3.1 Supported XML User-Defined Data Types

The following table lists the XML Schema data types supported by the `jwsc` and `wsdlc` Ant tasks and their equivalent Java data type or mapping mechanism.

For details and examples of the data types, see the JAX-RPC specification at `http://java.net/projects/jax-rpc/`.

Table 5–3 Supported User-Defined XML Schema Data Types

XML Schema Data Type	Equivalent Java Data Type or Mapping Mechanism
`<xsd:complexType>` with elements of both simple and complex types.	JavaBean
`<xsd:complexType>` with simple content.	JavaBean
`<xsd:attribute>` in `<xsd:complexType>`	Property of a JavaBean
Derivation of new simple types by restriction of an existing simple type.	Equivalent Java data type of simple type.
Facets used with restriction element.	Facets not enforced during serialization and deserialization.
`<xsd:list>`	Array of the list data type.
Array derived from `soapenc:Array` by restriction using the `wsdl:arrayType` attribute.	Array of the Java equivalent of the `arrayType` data type.
Array derived from `soapenc:Array` by restriction.	Array of Java equivalent.
Derivation of a complex type from a simple type.	JavaBean with a property called `_value` whose type is mapped from the simple type according to the rules in this section.
`<xsd:anyType>`	java.lang.Object
<xsd:any>	`javax.xml.soap.SOAPElement` or `org.apache.xmlbeans.XmlObject`
<xsd:any[]>	`javax.xml.soap.SOAPElement[]` or `org.apache.xmlbeans.XmlObject[]`
<xsd:union>	Common parent type of union members.
`<xsi:nil>` and `<xsd:nillable>` attribute	Java `null` value. If the XML data type is built-in and usually maps to a Java primitive data type (such as `int` or `short`), then the XML data type is actually mapped to the equivalent object wrapper type (such as `java.lang.Integer` or `java.lang.Short`).
Derivation of complex types	Mapped using Java inheritance.
Abstract types	Abstract Java data type.

5.3.2 Supported Java User-Defined Data Types

The following table lists the Java user-defined data types supported by the `jwsc` and `wsdlc` Ant tasks and their equivalent XML Schema data type.

Table 5–4 Supported User-Defined Java Data Types

Java Data Type	Equivalent XML Schema Data Type
JavaBean whose properties are any supported data type.	`<xsd:complexType>` whose content model is a `<xsd:sequence>` of elements corresponding to JavaBean properties.
Array and multidimensional array of any supported data type (when used as a JavaBean property)	An element in a `<xsd:complexType>` with the `maxOccurs` attribute set to `unbounded`.
`java.lang.Object` **Note:** The data type of the run-time object must be a known type.	`<xsd:anyType>`
Apache XMLBeans (that are inherited from `org.apache.xmlbeans.XmlObject` only) **Note**: The Web service that uses an Apache XMLBeans data type as a return type or parameter must be defined as `document-literal-wrapped` or `document-literal-bare`.	See Apache XMLBeans at `http://xmlbeans.apache.org/index.html`.
`java.util.Collection`	Literal Array
`java.util.List`	Literal Array
`java.util.ArrayList`	Literal Array
`java.util.LinkedList`	Literal Array
`java.util.Vector`	Literal Array
`java.util.Stack`	Literal Array
`java.util.Set`	Literal Array
`java.util.TreeSet`	Literal Array
`java.utils.SortedSet`	Literal Array
`java.utils.HashSet`	Literal Array

Note: The following user-defined Java data type, used as a parameter or return value of a WebLogic Web service in Version 8.1, is no longer supported: JAX-RPC-style enumeration class.

Additionally, generics are not supported when used as a parameter or return value. For example, the following Java method cannot be exposed as a public operation:

```
public ArrayList<String> echoGeneric(ArrayList<String> in) {
     return in;
 }
```

---

6 Developing JAX-RPC Web Service Clients

This chapter describes how to develop WebLogic Web service clients using Java API for XML-based RPC (JAX-RPC).

This chapter includes the following topics:

> **Note:** The following sections do not include information about invoking message-secured Web services; for that topic, see "Updating a Client Application to Invoke a Message-Secured Web Service" in *Securing WebLogic Web Services for Oracle WebLogic Server*.

6.1 Overview of JAX-RPC Web Service Clients

Invoking a Web service refers to the actions that a client application performs to use the Web service. Client applications that invoke Web services can be written using any technology: Java, Microsoft .NET, and so on.

There are two types of client applications:

- Java SE client—In its simplest form, a Java SE client is a Java program that has the `Main` public class that you invoke with the `java` command.
- Java EE component deployed to WebLogic Server—In this type of client application, the Web service runs inside a Java Platform, Enterprise Edition (Java EE) Version 5 component deployed to WebLogic Server, such as an EJB, servlet, or another Web service. This type of client application, therefore, runs inside a WebLogic Server container.

You can invoke a Web service from any Java SE or Java EE application running on WebLogic Server (with access to the WebLogic Server classpath). For information about support for *standalone* Java applications that are running in an environment

where WebLogic Server libraries are not available, see Section 6.4, "Using a Standalone Client JAR File When Invoking Web Services".

The sections that follow describe how to use Oracle's implementation of the JAX-RPC specification to invoke a Web service from a Java client application. You can use this implementation to invoke Web services running on any application server, both WebLogic and non-WebLogic. In addition, you can create a client that runs as part of a WebLogic Server, or a standalone client that runs in an environment where WebLogic Server libraries are not available.

In addition to the command-line tools described in this section, you can use an IDE, such as Oracle JDeveloper, for proxy generation and testing. For more information, see "Using Oracle IDEs to Build Web Services" in *Understanding WebLogic Web Services for Oracle WebLogic Server*.

Note: You cannot use a dynamic client to invoke a Web service operation that implements user-defined data types as parameters or return values. A dynamic client uses the JAX-RPC Call interface. Standard (static) clients use the Service and Stub JAX-RPC interfaces, which correctly invoke Web services that implement user-defined data types.

6.1.1 Invoking Web Services Using JAX-RPC

The Java API for XML based RPC (JAX-RPC) is a specification that defines the APIs used to invoke a Web service. WebLogic Server implements the JAX-RPC specification.

The following table briefly describes the core JAX-RPC interfaces and classes.

Table 6–1 JAX-RPC Interfaces and Classes

javax.xml.rpc Interface or Class	Description
`Service`	Main client interface.
`ServiceFactory`	Factory class for creating `Service` instances.
`Stub`	Base class of the client proxy used to invoke the operations of a Web service.
`Call`	Used to dynamically invoke a Web service.
`JAXRPCException`	Exception thrown if an error occurs while invoking a Web service.

6.1.2 Examples of Clients That Invoke Web Services

WebLogic Server optionally includes examples of creating and invoking WebLogic Web services in the *EXAMPLES_HOME*`/wl_server/examples/src/examples/webservices` directory, where *EXAMPLES_HOME* represents the directory in which the WebLogic Server code examples are configured. he default path is *ORACLE_HOME*`\user_projects\applications`. For detailed instructions on how to build and run the examples, open the *EXAMPLES_HOME*`\wl_server/docs/index.html` Web page in your browser and expand the **WebLogic Server Examples->Examples->API->Web Services** node. For more information, see "Sample Applications and Code Examples" in *Understanding Oracle WebLogic Server*.

6.2 Invoking a Web Service from a Java SE Client

> **Note:** As described in this section, you can invoke a Web service from any Java SE or Java EE application running on WebLogic Server (with access to the WebLogic Server classpath). For information about support for *standalone* Java applications that are running in an environment where WebLogic Server libraries are not available, see Section 6.4, "Using a Standalone Client JAR File When Invoking Web Services".

The following table summarizes the main steps to create a Java SE client that invokes a Web service.

> **Note:** It is assumed that you use Ant in your development environment to build your client application, compile Java files, and so on, and that you have an existing `build.xml` file that you want to update with Web services client tasks. For general information about using Ant in your development environment, see Section 3.5, "Creating the Basic Ant build.xml File." For a full example of a `build.xml` file used in this section, see Section 6.2.5, "Sample Ant Build File for a Java Client."

Table 6–2 Steps to Invoke a Web Service from a Java SE Client

#	Step	Description
1	Set up the environment.	Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) command, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`ORACLE_HOME`*`/user_projects/domains/`*`domainName`*, where *`ORACLE_HOME`* is the directory you specified as Oracle Home when you installed Oracle WebLogic Server and *`domainName`* is the name of your domain.
2	Update your `build.xml` file to execute the `clientgen` Ant task to generate the needed client-side artifacts to invoke a Web service.	See Section 6.2.1, "Using the clientgen Ant Task To Generate Client Artifacts."
3	Get information about the Web service, such as the signature of its operations and the name of the ports.	See Section 6.2.2, "Getting Information About a Web Service."
4	Write the client application Java code that includes code for invoking the Web service operation.	See Section 6.2.3, "Writing the Java Client Application Code to Invoke a Web Service."
5	Create a basic Ant build file, `build.xml`.	See Section 3.5, "Creating the Basic Ant build.xml File."
6	Compile and run your Java client application.	See Section 6.2.4, "Compiling and Running the Client Application."

6.2.1 Using the clientgen Ant Task To Generate Client Artifacts

The `clientgen` WebLogic Web services Ant task generates, from an existing WSDL file, the client artifacts that client applications use to invoke both WebLogic and non-WebLogic Web services. These artifacts include:

- The Java class for the JAX-RPC `Stub` and `Service` interface implementations for the particular Web service you want to invoke.
- The Java class for any user-defined XML Schema data types included in the WSDL file.
- The JAX-RPC mapping deployment descriptor file which contains information about the mapping between the Java user-defined data types and their corresponding XML Schema types in the WSDL file.
- A client-side copy of the WSDL file.

For additional information about the `clientgen` Ant task, such as all the available attributes, see "Ant Task Reference" in the *WebLogic Web Services Reference for Oracle WebLogic Server*.

Update your `build.xml` file, adding a call to the `clientgen` Ant task, as shown in the following example:

```
<taskdef name="clientgen"
    classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
 <target name="build-client">
    <clientgen
      wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
      destDir="clientclasses"
      packageName="examples.webservices.simple_client"
      type="JAXRPC"/>
 </target>
```

Before you can execute the `clientgen` WebLogic Web service Ant task, you must specify its full Java classname using the standard `taskdef` Ant task.

You must include the `wsdl` and `destDir` attributes of the `clientgen` Ant task to specify the WSDL file from which you want to create client-side artifacts and the directory into which these artifacts should be generated. The `packageName` attribute is optional; if you do not specify it, the `clientgen` task uses a package name based on the `targetNamespace` of the WSDL. The `type` is also optional; if not specified, it defaults to `JAXRPC`.

In this example, the package name is set to the same package name as the client application, `examples.webservices.simple_client`. If you set the package name to one that is different from the client application, you would need to import the appropriate class files. For example, if you defined the package name as `examples.webservices.complex`, you would need to import the following class files in the client application:

```
import examples.webservices.complex.BasicStruct;
 import examples.webservices.complex.ComplexPortType;
 import examples.webservices.complex.ComplexService;
```

> **Note:** The `clientgen` Ant task also provides the `destFile` attribute if you want the Ant task to automatically compile the generated Java code and package all artifacts into a JAR file. For details and an example, see "clientgen" in the *WebLogic Web Services Reference for Oracle WebLogic Server*.

If the WSDL file specifies that user-defined data types are used as input parameters or return values of Web service operations, `clientgen` automatically generates a JavaBean class that is the Java representation of the XML Schema data type defined in the WSDL. The JavaBean classes are generated into the `destDir` directory.

> **Note:** The package of the Java user-defined data type is based on the XML Schema of the data type in the WSDL, which is different from the package name of the JAX-RPC stubs.

For a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`, see Section 6.2.5, "Sample Ant Build File for a Java Client."

To execute the `clientgen` Ant task, along with the other supporting Ant tasks, specify the `build-client` target at the command line:

```
prompt> ant build-client
```

See the `clientclasses` directory to view the files and artifacts generated by the `clientgen` Ant task.

6.2.2 Getting Information About a Web Service

You need to know the name of the Web service and the signature of its operations before you write your Java client application code to invoke an operation. There are a variety of ways to find this information.

The best way to get this information is to use the `clientgen` Ant task to generate the Web service-specific JAX-RPC stubs and look at the generated `*.java` files. These files are generated into the directory specified by the `destDir` attribute, with subdirectories corresponding to either the value of the `packageName` attribute, or, if this attribute is not specified, to a package based on the `targetNamespace` of the WSDL.

- The *`ServiceName`*`.java` source file contains the `getPortName()` methods for getting the Web service port, where *`ServiceName`* refers to the name of the Web service and *`PortName`* refers to the name of the port. If the Web service was implemented with a JWS file, the name of the Web service is the value of the `serviceName` attribute of the `@WebService` JWS annotation and the name of the port is the value of the `portName` attribute of the `@WLHttpTransport` annotation.
- The *`PortType`*`.java` file contains the method signatures that correspond to the public operations of the Web service, where *`PortType`* refers to the port type of the Web service. If the Web service was implemented with a JWS file, the port type is the value of the `name` attribute of the `@WebService` JWS annotation.

You can also examine the actual WSDL of the Web service; see Section 3.10, "Browsing to the WSDL of the Web Service" for details about the WSDL of a deployed WebLogic

Web service. The name of the Web service is contained in the `<service>` element, as shown in the following excerpt of the `TraderService` WSDL:

```
<service name="TraderService">
   <port name="TraderServicePort"
        binding="tns:TraderServiceSoapBinding">
 ...
   </port>
 </service>
```

The operations defined for this Web service are listed under the corresponding `<binding>` element. For example, the following WSDL excerpt shows that the `TraderService` Web service has two operations, `buy` and `sell` (for clarity, only relevant parts of the WSDL are shown):

```
<binding name="TraderServiceSoapBinding" ...>
   ...
   <operation name="sell">
   ...
   </operation>
   <operation name="buy">
   </operation>
 </binding>
```

6.2.3 Writing the Java Client Application Code to Invoke a Web Service

In the following code example, a Java application invokes a Web service operation. The client application takes a single argument: the WSDL of the Web service.The application then uses standard JAX-RPC API code and the Web service-specific implementation of the `Service` interface, generated by `clientgen`, to invoke an operation of the Web service.

The example also shows how to invoke an operation that has a user-defined data type (`examples.webservices.complex.BasicStruct`) as an input parameter and return value. The `clientgen` Ant task automatically generates the Java code for this user-defined data type.

```
package examples.webservices.simple_client;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
// import the BasicStruct class, used as a param and return value of the
 // echoComplexType operation.  The class is generated automatically by
 // the clientgen Ant task.
import examples.webservices.complex.BasicStruct;
/**
  * This is a simple Java client application that invokes the
  * the echoComplexType operation of the ComplexService Web service.
  */
public class Main {
  public static void main(String[] args)
       throws ServiceException, RemoteException {
    ComplexService service = new ComplexService_Impl (args[0] + "?WSDL" );
     ComplexPortType port = service.getComplexServicePort();
    BasicStruct in = new BasicStruct();
    in.setIntValue(999);
     in.setStringValue("Hello Struct");
    BasicStruct result = port.echoComplexType(in);
     System.out.println("echoComplexType called. Result: " + result.getIntValue()
```

```
+ ", " + result.getStringValue());
   }
 }
```

In the preceding example:

- The following code shows how to create a `ComplexPortType` stub:

  ```
  ComplexService service = new ComplexService_Impl (args[0] + "?WSDL");
   ComplexPortType port = service.getComplexServicePort();
  ```

  The `ComplexService_Impl` stub factory implements the JAX-RPC `Service` interface. The constructor of `ComplexService_Impl` creates a stub based on the provided WSDL URI (`args[0] + "?WSDL"`). The `getComplexServicePort()` method is used to return an instance of the `ComplexPortType` stub implementation.

- The following code shows how to invoke the `echoComplexType` operation of the `ComplexService` Web service:

  ```
  BasicStruct result = port.echoComplexType(in);
  ```

  The `echoComplexType` operation returns the user-defined data type called `BasicStruct`.

The method of your application that invokes the Web service operation must throw or catch `java.rmi.RemoteException` and `javax.xml.rpc.ServiceException`, both of which are thrown from the generated JAX-RPC stubs.

6.2.4 Compiling and Running the Client Application

Add `javac` tasks to the `build-client` target in the `build.xml` file to compile all the Java files (both of your client application and those generated by `clientgen`) into class files, as shown by the **bold** text in the following example:

```
<target name="build-client">
  <clientgen
     wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
     destDir="clientclasses"
     packageName="examples.webservices.simple_client"
     type="JAXRPC"/>
  <javac
     srcdir="clientclasses"
     destdir="clientclasses"
     includes="**/*.java"/>
  <javac
     srcdir="src"
     destdir="clientclasses"
     includes="examples/webservices/simple_client/*.java"/>
</target>
```

In the example, the first `javac` task compiles the Java files in the `clientclasses` directory that were generated by `clientgen`, and the second `javac` task compiles the Java files in the `examples/webservices/simple_client` subdirectory of the current directory; where it is assumed your Java client application source is located.

In the preceding example, the `clientgen`-generated Java source files and the resulting compiled classes end up in the same directory (`clientclasses`). Although this might be adequate for prototyping, it is often a best practice to keep source code (even generated code) in a different directory from the compiled classes. To do this, set the `destdir` for both `javac` tasks to a directory different from the `srcdir` directory.

You must also copy the following `clientgen`-generated files from `clientgen`'s destination directory to `javac`'s destination directory, keeping the same subdirectory hierarchy in the destination:

packageName/*ServiceName*_internaldd.xml
packageName/*ServiceName*_java_wsdl_mapping.xml
packageName/*ServiceName*_saved_wsdl.wsdl

where *packageName* refers to the subdirectory hierarchy that corresponds to the package of the generated JAX-RPC stubs and *ServiceName* refers to the name of the Web service.

To run the client application, add a `run` target to the `build.xml` that includes a call to the `java` task, as shown below:

```
<path id="client.class.path">
     <pathelement path="clientclasses"/>
     <pathelement path="${java.class.path}"/>
 </path>
<target name="run" >
     <java
        fork="true"
        classname="examples.webServices.simple_client.Main"
        failonerror="true" >
        <classpath refid="client.class.path"/>
   <arg line="http://${wls.hostname}:${wls.port}/complex/ComplexService" />
 </target>
```

The `path` task adds the `clientclasses` directory to the CLASSPATH. The `run` target invokes the `Main` application, passing it the URL of the deployed Web service as its single argument.

See Section 6.2.5, "Sample Ant Build File for a Java Client" for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`.

Rerun the `build-client` target to regenerate the artifacts and recompile into classes, then execute the `run` target to invoke the `echoStruct` operation:

```
prompt> ant build-client run
```

You can use the `build-client` and `run` targets in the `build.xml` file to iteratively update, rebuild, and run the Java client application as part of your development process.

6.2.5 Sample Ant Build File for a Java Client

The following example shows a complete `build.xml` file for generating and compiling a Java client. See Section 6.2.1, "Using the clientgen Ant Task To Generate Client Artifacts" and Section 6.2.4, "Compiling and Running the Client Application" for explanations of the sections in **bold**.

```
<project name="webservices-simple_client" default="all">
  <!-- set global properties for this build -->
  <property name="wls.hostname" value="localhost" />
   <property name="wls.port" value="7001" />
  <property name="example-output" value="output" />
   <property name="clientclass-dir" value="${example-output}/clientclass" />
  <path id="client.class.path">
     <pathelement path="${clientclass-dir}"/>
     <pathelement path="${java.class.path}"/>
```

```
    </path>
  <taskdef name="clientgen"
     classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
  <target name="clean" >
     <delete dir="${clientclass-dir}"/>
   </target>
  <target name="all" depends="clean,build-client,run" />
  <target name="build-client">
    <clientgen
       wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
       destDir="${clientclass-dir}"
       packageName="examples.webservices.simple_client"
       type="JAXRPC"/>
    <javac
       srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
       includes="**/*.java"/>
    <javac
       srcdir="src" destdir="${clientclass-dir}"
       includes="examples/webservices/simple_client/*.java"/>
   </target>
  <target name="run" >
     <java fork="true"
           classname="examples.webservices.simple_client.Main"
           failonerror="true" >
       <classpath refid="client.class.path"/>
       <arg  line="http://${wls.hostname}:${wls.port}/complex/ComplexService"
       />
     </java>
   </target>
</project>
```

6.3 Invoking a Web Service from Another Web Service

Invoking a Web service from within a WebLogic Web service is similar to invoking one from another Java application, as described in Section 6.2, "Invoking a Web Service from a Java SE Client." However, instead of using the `clientgen` Ant task to generate the JAX-RPC stubs of the Web service to be invoked, you use the `<clientgen>` child element of the `<jws>` element, inside the `jwsc` Ant task that compiles the invoking Web service. In the JWS file that invokes the other Web service, however, you still use the same standard JAX-RPC APIs to get `Service` and `PortType` instances to invoke the Web service operations.

It is assumed that you have read and understood Section 6.2, "Invoking a Web Service from a Java SE Client." It is also assumed that you use Ant in your development environment to build your client application, compile Java files, and so on, and that you have an existing `build.xml` that builds a Web service that you want to update to invoke another Web service.

The following list describes the changes you must make to the `build.xml` file that builds your client Web service, which will invoke another Web service. See Section 6.3.1, "Sample build.xml File for a Web Service Client" for the full sample `build.xml` file:

- Add a `<clientgen>` child element to the `<jws>` element that specifies the JWS file that implements the Web service that invokes another Web service. Set the required `wsdl` attribute to the WSDL of the Web service to be invoked. Set the required `packageName` attribute to the package into which you want the JAX-RPC client stubs to be generated.

The following list describes the changes you must make to the JWS file that implements the client Web service; see Section 6.3.2, "Sample JWS File That Invokes a Web Service" for the full JWS file example.

- Import the files generated by the `<clientgen>` child element of the `jwsc` Ant task. These include the JAX-RPC stubs of the invoked Web service, as well as the Java representation of any user-defined data types used as parameters or return values in the operations of the invoked Web service.

  Note: The user-defined data types are generated into a package based on the XML Schema of the data type in the WSDL, *not* in the package specified by `clientgen`. The JAX-RPC stubs, however, use the package name specified by the `packageName` attribute of the `<clientgen>` element.

- Update the method that contains the invoke of the Web service to either throw or catch both `java.rmi.RemoteException` and `javax.xml.rpc.ServiceException`.
- Get the `Service` and `PortType` JAX-RPC stubs implementation and invoke the operation on the port as usual; see Section 6.2.3, "Writing the Java Client Application Code to Invoke a Web Service" for details.

6.3.1 Sample build.xml File for a Web Service Client

The following sample `build.xml` file shows how to create a Web service that itself invokes another Web service; the relevant sections that differ from the `build.xml` for building a simple Web service that does not invoke another Web service are shown in **bold**.

The `build-service` target in this case is very similar to a target that builds a simple Web service; the only difference is that the `jwsc` Ant task that builds the invoking Web service also includes a `<clientgen>` child element of the `<jws>` element so that `jwsc` also generates the required JAX-RPC client stubs.

```
<project name="webservices-service_to_service" default="all">
  <!-- set global properties for this build -->
  <property name="wls.username" value="weblogic" />
   <property name="wls.password" value="weblogic" />
   <property name="wls.hostname" value="localhost" />
   <property name="wls.port" value="7001" />
   <property name="wls.server.name" value="myserver" />
  <property name="ear.deployed.name" value="ClientServiceEar" />
   <property name="example-output" value="output" />
   <property name="ear-dir" value="${example-output}/ClientServiceEar" />
   <property name="clientclass-dir" value="${example-output}/clientclasses" />
  <path id="client.class.path">
     <pathelement path="${clientclass-dir}"/>
     <pathelement path="${java.class.path}"/>
   </path>
  <taskdef name="jwsc"
     classname="weblogic.wsee.tools.anttasks.JwscTask" />
  <taskdef name="clientgen"
     classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
  <taskdef name="wldeploy"
     classname="weblogic.ant.taskdefs.management.WLDeploy"/>
  <target name="all" depends="clean,build-service,deploy,client" />
  <target name="clean" depends="undeploy">
```

```
    <delete dir="${example-output}"/>
   </target>
  <target name="build-service">
    <jwsc
         srcdir="src"
         destdir="${ear-dir}" >
        <jws
          file="examples/webservices/service_to_service/ClientServiceImpl.java"
          type="JAXRPC">
           <clientgen

wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
                 packageName="examples.webservices.complex" />
         </jws>
    </jwsc>
  </target>
  <target name="deploy">
     <wldeploy action="deploy" name="${ear.deployed.name}"
       source="${ear-dir}" user="${wls.username}"
       password="${wls.password}" verbose="true"
       adminurl="t3://${wls.hostname}:${wls.port}"
       targets="${wls.server.name}" />
   </target>
  <target name="undeploy">
     <wldeploy action="undeploy" name="${ear.deployed.name}"
       failonerror="false"
       user="${wls.username}"
       password="${wls.password}" verbose="true"
       adminurl="t3://${wls.hostname}:${wls.port}"
       targets="${wls.server.name}" />
   </target>
  <target name="client">
    <clientgen
       wsdl="http://${wls.hostname}:${wls.port}/ClientService/ClientService?WSDL"
       destDir="${clientclass-dir}"
       packageName="examples.webservices.service_to_service.client"
       type="JAXRPC"/>
    <javac
       srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
       includes="**/*.java"/>
    <javac
       srcdir="src" destdir="${clientclass-dir}"
       includes="examples/webservices/service_to_service/client/**/*.java"/>
  </target>
  <target name="run">
     <java classname="examples.webservices.service_to_service.client.Main"
           fork="true"
           failonerror="true" >
           <classpath refid="client.class.path"/>
           <arg

line="http://${wls.hostname}:${wls.port}/ClientService/ClientService"/>
    </java>
  </target>
</project>
```

6.3.2 Sample JWS File That Invokes a Web Service

The following sample JWS file, called `ClientServiceImpl.java`, implements a Web service called `ClientService` that has an operation that in turn invokes the `echoComplexType` operation of a Web service called `ComplexService`. This operation has a user-defined data type (`BasicStruct`) as both a parameter and a return value. The relevant code is shown in **bold** and described after the example.

```
package examples.webservices.service_to_service;
import java.rmi.RemoteException;
 import javax.xml.rpc.ServiceException;
import javax.jws.WebService;
 import javax.jws.WebMethod;
import weblogic.jws.WLHttpTransport;
// Import the BasicStruct data type, generated by clientgen and used
 // by the ComplexService Web Service
import examples.webservices.complex.BasicStruct;
// Import the JAX-RPC Stubs for invoking the ComplexService Web Service.
 // Stubs generated by clientgen
import examples.webservices.service_to_service.ComplexPortType;
 import examples.webservices.service_to_service.ComplexService_Impl;
 import examples.webservices.service_to_service.ComplexService;
@WebService(name="ClientPortType", serviceName="ClientService",
             targetNamespace="http://examples.org")
@WLHttpTransport(contextPath="ClientService", serviceUri="ClientService",
                  portName="ClientServicePort")
public class ClientServiceImpl {
  @WebMethod()
   public String callComplexService(BasicStruct input, String serviceUrl)
       throws ServiceException, RemoteException
   {
    // Create service and port stubs to invoke ComplexService
     ComplexService service = new ComplexService_Impl(serviceUrl + "?WSDL");
     ComplexPortType port = service.getComplexServicePort();
    // Create service and port stubs to invoke ComplexService
     ComplexService service = new ComplexService_Impl(serviceUrl + "?WSDL");
     ComplexPortType port = service.getComplexServicePortTypePort();
     // Invoke the echoComplexType operation of ComplexService
     BasicStruct result = port.echoComplexType(input);
     System.out.println("Invoked ComplexPortType.echoComplexType." );
    return "Invoke went okay!  Here's the result: '" + result.getIntValue() + ",
" + result.getStringValue() + "'";
  }
 }
```

Follow these guidelines when programming the JWS file that invokes another Web service; code snippets of the guidelines are shown in **bold** in the preceding example:

- Import any user-defined data types that are used by the invoked Web service. In this example, the `ComplexService` uses the `BasicStruct` JavaBean:

  ```
  import examples.webservices.complex.BasicStruct;
  ```

- Import the JAX-RPC stubs of the `ComplexService` Web service; the stubs are generated by the `<cliengen>` child element of `<jws>`:

  ```
  import examples.webservices.service_to_service.ComplexPortType;
   import examples.webservices.service_to_service.ComplexService_Impl;
   import examples.webservices.service_to_service.ComplexService;
  ```

- Ensure that your client Web service throws or catches `ServiceException` and `RemoteException`:

```
throws ServiceException, RemoteException
```

- Create the JAX-RPC `Service` and `PortType` instances for the `ComplexService`:

```
ComplexService service = new
      ComplexService_Impl(serviceUrl + "?WSDL");
 ComplexPortType port = service.getComplexServicePortTypePort();
```

- Invoke the `echoComplexType` operation of `ComplexService` using the port you just instantiated:

```
BasicStruct result = port.echoComplexType(input);
```

6.4 Using a Standalone Client JAR File When Invoking Web Services

It is assumed in this document that, when you invoke a Web service using the client-side artifacts generated by the `clientgen` or `wsdlc` Ant tasks, you have the entire set of WebLogic Server classes in your CLASSPATH. If, however, your computer does *not* have WebLogic Server installed, you can still invoke a Web service by using the standalone WebLogic Web services client JAR file, as described in this section.

The standalone client JAR file supports basic client-side functionality, such as:

- Use with client-side artifacts created by both the `clientgen` Ant tasks
- Processing SOAP messages
- Using client-side SOAP message handlers
- Using MTOM
- Invoking JAX-RPC Web services
- Using SSL

The standalone client JAR file does *not*, however, support invoking Web services that use the following advanced features:

- Web services reliable SOAP messaging
- Message-level security (WS-Security)
- Conversations
- Asynchronous request-response
- Buffering
- JMS transport

To use the standalone WebLogic Web services client JAR file with your client application, follow these steps:

1. Copy the file *`ORACLE_HOME`*`/wlserver/modules/clients/com.oracle.webservices.wls.jaxrpc-client_12.1.2.jar` from the computer hosting WebLogic Server to the client computer, where *`ORACLE_HOME`* is the directory you specified as Oracle Home when you installed Oracle WebLogic Server and domainName is the name of your domain.
2. Add the JAR file to your CLASSPATH.

Note: Ensure that your CLASSPATH includes the JAR file that contains the Ant classes (`ant.jar`) as a subset of the classes are required by the JAR file. This JAR file is typically located in the `lib` directory of the Ant distribution.

6.5 Using a Proxy Server When Invoking a Web Service

You can use a proxy server to proxy requests from a client application to an application server (either WebLogic or non-WebLogic) that hosts the invoked Web service. You typically use a proxy server when the application server is behind a firewall. There are two ways to specify the proxy server in your client application: programmatically using the WebLogic `HttpTransportInfo` API or using system properties.

6.5.1 Using the HttpTransportInfo API to Specify the Proxy Server

You can programmatically specify within the Java client application itself the details of the proxy server that will proxy the Web service invoke by using the standard `java.net.*` classes and the WebLogic-specific `HttpTransportInfo` API. You use the `java.net` classes to create a `Proxy` object that represents the proxy server, and then use the WebLogic API and properties to set the proxy server on the JAX-RPC stub, as shown in the following sample client that invokes the `echo` operation of the `HttpProxySampleService` Web service. The code in **bold** is described after the example:

```
package dev2dev.proxy.client;
import java.net.Proxy;
 import java.net.InetSocketAddress;
import weblogic.wsee.connection.transport.http.HttpTransportInfo;
/**
  * Sample client to invoke a service through a proxy server via
  * programmatic API
  */
public class HttpProxySampleClient {
   public static void main(String[] args) throws Throwable{
     assert args.length == 5;
     String endpoint = args[0];
     String proxyHost = args[1];
     String proxyPort = args[2];
     String user = args[3];
     String pass = args[4];
    //create service and port
     HttpProxySampleService service = new HttpProxySampleService_Impl();
     HttpProxySamplePortType port = service.getHttpProxySamplePortTypeSoapPort();
    //set endpoint address
     ((Stub)port)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY, endpoint);
    //set proxy server info
    Proxy p = new Proxy(Proxy.Type.HTTP, new  InetSocketAddress(proxyHost,
Integer.parseInt(proxyPort)));
     HttpTransportInfo info = new HttpTransportInfo();
     info.setProxy(p);
     ((Stub)port)._setProperty("weblogic.wsee.connection.transportinfo",info);
    //set proxy-authentication info
     ((Stub)port)._setProperty("weblogic.webservice.client.proxyusername",user);
     ((Stub)port)._setProperty("weblogic.webservice.client.proxypassword",pass);
    //invoke
     String s = port.echo("Hello World!");
```

```
    System.out.println("echo: " + s);
  }
}
```

The sections of the preceding example to note are as follows:

- Import the required `java.net.*` classes:

  ```
  import java.net.Proxy;
   import java.net.InetSocketAddress;
  ```

- Import the WebLogic `HttpTransportInfo` API:

  ```
  import weblogic.wsee.connection.transport.http.HttpTransportInfo;
  ```

- Create a `Proxy` object that represents the proxy server:

  ```
  Proxy p = new Proxy(Proxy.Type.HTTP, new  InetSocketAddress(proxyHost,
  Integer.parseInt(proxyPort)));
  ```

  The `proxyHost` and `proxyPort` arguments refer to the host computer and port of the proxy server.

- Create an `HttpTransportInfo` object and use the setProxy() method to set the proxy server information:

  ```
  HttpTransportInfo info = new HttpTransportInfo();
   info.setProxy(p);
  ```

- Use the `weblogic.wsee.connection.transportinfo` WebLogic stub property to set the `HttpTransportInfo` object on the JAX-RPC stub:

  ```
  ((Stub)port)._setProperty("weblogic.wsee.connection.transportinfo",info);
  ```

- Use `weblogic.webservice.client.proxyusername` and `weblogic.webservice.client.proxypassword` WebLogic-specific stub properties to specify the username and password of a user who is authenticated to access the proxy server:

  ```
  ((Stub)port)._setProperty("weblogic.webservice.client.proxyusername",user);
       ((Stub)port)._
  setProperty("weblogic.webservice.client.proxypassword",pass);
  ```

  Alternatively, you can use the `setProxyUsername()` and `setProxyPassword()` methods of the `HttpTransportInfo` API to set the proxy username and password, as shown in the following example:

  ```
      info.setProxyUsername("juliet".getBytes());
       info.setProxyPassword("secret".getBytes());
  ```

6.5.2 Using System Properties to Specify the Proxy Server

To use system properties to specify the proxy server, write your client application in the standard way, and then specify system properties when you execute the client application.

You have a choice of using standard Java system properties or historical WebLogic properties. If the proxySet system property is set to `false` (proxySet=false), proxy properties will be ignored and no proxy will be used.

The following table summarizes the Java system properties. In this case, the proxySet system property must not be set.

Table 6–3 Java System Properties Used to Specify Proxy Server

Property	Description
http.proxyHost=*proxyHost* or https.proxyHost=*proxyHost*	Name of the host computer on which the proxy server is running. Use https.proxyHost for HTTP over SSL.
http.proxyPort=*proxyPort* or https.proxy.Port=*proxyPort*	Port to which the proxy server is listening. Use https.proxyPort for HTTP over SSL.
http.nonProxyHosts= *hostname* | *hostname* | ...	List of hosts that should be reached directly, bypassing the proxy. Separate each host name using a | character. This property applies to both HTTP and HTTPS.

The following excerpt from an Ant build script shows an example of setting Java system properties when invoking a client application called `clients.InvokeMyService`:

```
<target name="run-client">
   <java fork="true"
         classname="clients.InvokeMyService"
         failonerror="true">
     <classpath refid="client.class.path"/>
     <arg line="${http-endpoint}"/>
     <jvmarg line=
       "-Dhttp.proxyHost=${proxy-host}
       -Dhttp.proxyPort=${proxy-port}
       -Dhttp.nonProxyHosts=${mydomain}"
     />
   </java>
 </target>
```

The following table summarizes the WebLogic system properties. In this case, the proxySet system property must be set to `true`.

Table 6–4 WebLogic System Properties Used to Specify the Proxy Server

Property	Description
proxySet=true	Flag that specifies that the historical WebLogic proxy properties should be used.
proxyHost=*proxyHost*	Name of the host computer on which the proxy server is running.
proxyPort=*proxyPort*	Port to which the proxy server is listening.
weblogic.webservice.client. proxyusername=*username*	Username used to access the proxy server.
weblogic.webservice.client. proxypassword=*password*	Password used to access the proxy server.

The following excerpt from an Ant build script shows an example of setting WebLogic system properties when invoking a client application called `clients.InvokeMyService`:

```
<target name="run-client">
   <java fork="true"
         classname="clients.InvokeMyService"
```

```
        failonerror="true">
      <classpath refid="client.class.path"/>
      <arg line="${http-endpoint}"/>
      <jvmarg line=
       "-DproxySet=true
        -DproxyHost=${proxy-host}
        -DproxyPort=${proxy-port}
        -Dweblogic.webservice.client.proxyusername=${proxy-username}
        -Dweblogic.webservice.client.proxypassword=${proxy-passwd}"
      />
    </java>
  </target>
```

6.6 Client Considerations When Redeploying a Web Service

WebLogic Server supports production redeployment, which means that you can deploy a new version of an updated WebLogic Web service alongside an older version of the same Web service.

WebLogic Server automatically manages client connections so that only *new* client requests are directed to the new version. Clients already connected to the Web service during the redeployment continue to use the older version of the service until they complete their work, at which point WebLogic Server automatically retires the older Web service. If the client is connected to a conversational or reliable Web service, its work is considered complete when the existing conversation or reliable messaging sequence is explicitly ended by the client or because of a timeout.

You can continue using the old client application with the new version of the Web service, as long as the following Web service artifacts have not changed in the new version:

- WSDL that describes the Web service
- WS-Policy files attached to the Web service

If any of these artifacts have changed, you must regenerate the JAX-RPC stubs used by the client application by re-running the `clientgen` Ant task.

For example, if you change the signature of an operation in the new version of the Web service, then the WSDL file that describes the new version of the Web service will also change. In this case, you must regenerate the JAX-RPC stubs. If, however, you simply change the implementation of an operation, but do not change its public contract, then you can continue using the existing client application.

6.7 WebLogic Web Services Stub Properties

WebLogic Server provides a set of stub properties that you can set in the JAX-RPC Stub used to invoke a WebLogic Web service. Use the `Stub._setProperty()` method to set the properties, as shown in the following example:

```
((Stub)port)._setProperty(WLStub.MARSHAL_FORCE_INCLUDE_XSI_TYPE,"true");
```

Most of the stub properties are defined in the `WLStub` class. See `"weblogic.wsee.jaxrpc.WLStub"` in the *Java API Reference for Oracle WebLogic Server*for details.

The following table describes additional stub properties not defined in the `WLStub` class.

Table 6–5 Additional Stub Properties

Stub Property	Description
`weblogic.wsee.transport.connection.timeout`	Specifies, in milliseconds, how long a client application that is attempting to invoke a Web service waits to make a connection. After the specified time elapses, if a connection hasn't been made, the attempt times out.
`weblogic.wsee.transport.read.timeout`	Specifies, in milliseconds, how long a client application waits for a response from a Web service it is invoking. After the specified time elapses, if a response hasn't arrived, the client times out.
`weblogic.wsee.security.bst.serverVerifyCert`	Specifies the certificate that the client application uses to validate the signed response from WebLogic Server. By default, WebLogic Server includes the certification used to validate in the response SOAP message itself; if this is not possible, then use this stub property to specify a different one. This stub property applies *only* to client applications that run inside of a WebLogic Server container, and not to standalone client applications. The value of the property is an object of data type `java.security.cert.X509Certificate`.
`weblogic.wsee.security.bst.serverEncryptCert`	Specifies the certificate that the client application uses to encrypt the request SOAP message sent to WebLogic Server. By default, the client application uses the public certificate published in the Web service's WSDL; if this is not possible, then use this stub property to specify a different one. This stub property applies *only* to client applications that run inside of a WebLogic Server container, and not to standalone client applications. The value of the property is an object of data type `java.security.cert.X509Certificate`.

`weblogic.wsee.marshal.forceIncludeXsiType`	Specifies that the SOAP messages for a Web service operation invoke should include the XML Schema data type of each parameter. By default, the SOAP messages do not include the data type of each parameter.
	If you set this property to `True`, the elements in the SOAP messages that describe operation parameters will include an `xsi:type` attribute to specify the data type of the parameter, as shown in the following example:
	`<soapenv:Envelope>`
	`...`
	`<maxResults xsi:type="xs:int">10</maxResults>`
	`...`
	By default (or if you set this property to `False`), the parameter element would look like the following example:
	`<soapenv:Envelope>`
	`...`
	`<maxResults>10</maxResults>`
	`...`
	Valid values for this property are `True` and `False`; default value is `False`.

6.8 Setting the Character Encoding For the Response SOAP Message

Use the `weblogic.wsee.jaxrpc.WLStub.CHARACTER_SET_ENCODING` WLStub property to set the character encoding of the response (outbound) SOAP message. You can set it to the following two values:

- UTF-8
- UTF-16

The following code snippet from a client application shows how to set the character encoding to UTF-16:

```
  Simple port = service.getSimpleSoapPort();
   ((Stub) port)._setProperty(weblogic.wsee.jaxrpc.WLStub.CHARACTER_SET_ENCODING,
"UTF-16");
   port.invokeMethod();
```

See `"weblogic.wsee.jaxrpc.WLStub"` in the *Java API Reference for Oracle WebLogic Server* for additional WLStub properties you can set.

Part III

Developing Advanced Features of JAX-RPC Web Services

Part II describes how to develop advanced features of WebLogic Web services using Java API for XML-based RPC (JAX-RPC).

Sections include:

7

Invoking a Web Service Using Asynchronous Request-Response

This chapter describes how to invoke a WebLogic Java API for XML-based RPC (JAX-RPC) Web service using asynchronous request-response.

This chapter includes the following topics:

7.1 Overview of the Asynchronous Request-Response Feature

When you invoke a Web service synchronously, the invoking client application waits for the response to return before it can continue with its work. In cases where the response returns immediately, this method of invoking the Web service might be adequate. However, because request processing can be delayed, it is often useful for the client application to continue its work and handle the response later on, or in other words, use the asynchronous request-response feature of WebLogic Web services.

You invoke a Web service asynchronously only from a client running in a WebLogic Web service, never from a stand-alone client application. The invoked Web service does not change in any way, thus you can invoke any deployed Web service (both WebLogic and non-WebLogic) asynchronously as long as the application server that hosts the Web service supports the WS-Addressing specification at `http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/`.

When implementing asynchronous request-response in your client, rather than invoking the operation directly, you invoke an asynchronous flavor of the same operation. (This asynchronous flavor of the operation is automatically generated by the `jwsc` Ant task.) For example, rather than invoking an operation called `getQuote` directly, you would invoke `getQuoteAsync` instead. The asynchronous flavor of the operation always returns `void`, even if the original operation returns a value. You then include methods in your client that handle the asynchronous response or failures

when it returns later on. You put any business logic that processes the return value of the Web service operation invoke or a potential failure in these methods. You use both naming conventions and JWS annotations to specify these methods to the JWS compiler. For example, if the asynchronous operation is called `getQuoteAsync`, then these methods might be called `onGetQuoteAsyncResponse` and `onGetQuoteAsyncFailure`.

Note: For information about using asynchronous request-response with other asynchronous features, such as Web service reliable messaging or buffering, see Chapter 11, "Using the Asynchronous Features Together." This section describes how to use the asynchronous request-response feature on its own.

The asynchronous request-response feature works only with HTTP; you cannot use it with the HTTPS or JMS transport.

7.2 Using Asynchronous Request-Response: Main Steps

The following procedure describes how to create a client Web service that asynchronously invokes an operation in a different Web service. The procedure shows how to create the JWS file that implements the client Web service from scratch; if you want to update an existing JWS file, use this procedure as a guide.

For clarity, it is assumed in the procedure that:

- The client Web service is called `StockQuoteClientService`.
- The `StockQuoteClientService` service is going to invoke the `getQuote(String)` operation of the already-deployed `StockQuoteService` service whose WSDL is found at the following URL:

  ```
  http://localhost:7001/async/StockQuote?WSDL
  ```

It is further assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file to which you can add targets for running the `jwsc` Ant task and deploying the generated service. For more information, see the following sections:

- "Examples for JAX-RPC Web Service Developers" on page 2-1
- "Developing JAX-RPC Web Services" on page 3-1
- "Programming the JWS File" on page 4-1
- "Developing JAX-RPC Web Service Clients" on page 6-1

Table 7–1 Steps to Use Asynchronous Request-Response

#	Step	Description
1	Configure the WebLogic Server instances.	Configure the asynchronous response service, as described in Section 7.3, "Configuring the Host WebLogic Server Instance for the Asynchronous Web Service".
2	Create a new JWS file, or update an existing one, that implements the `StockQuoteClientService` Web service.	Use your favorite IDE or text editor. See Section 7.4, "Writing the Asynchronous JWS File".

Table 7–1 (Cont.) Steps to Use Asynchronous Request-Response

#	Step	Description
3	Update your `build.xml` file to compile the JWS file that implements the `StockQuoteClientService`.	You will add a `<clientgen>` child element to the `jwsc` Ant task so as to automatically generate the asynchronous flavor of the Web service operations you are invoking. See Section 7.5, "Updating the build.xml File When Using Asynchronous Request-Response".
4	Run the Ant target to build the `StockQuoteClientService`.	For example: `prompt> ant build-clientService`
5	Deploy the `StockQuoteClientService` Web service as usual.	See "Deploying and Undeploying WebLogic Web Services" on page 3-14.

When you invoke the `StockQuoteClientService` Web service, which in turn invokes the `StockQuoteService` Web service, the second invoke will be asynchronous rather than synchronous.

7.3 Configuring the Host WebLogic Server Instance for the Asynchronous Web Service

Configuring the WebLogic Server instance on which the asynchronous Web service is deployed involves configuring JMS resources, such as JMS servers and modules, that are used internally by the Web services runtime.

You can configure these resources manually or you can use the Configuration Wizard to extend the WebLogic Server domain using a Web services-specific extension template. Using the Configuration Wizard greatly simplifies the required configuration steps; for details, see "Configuring Your Domain For Web Services Features" on page 3-2.

Notes: Alternatively, you can use WLST to configure the resources. For information about using WLST to extend the domain, see "Configuring Existing Domains" in *Understanding the WebLogic Scripting Tool*.

A domain that does not contain Web Services resources will still boot and operate correctly for non-Web services scenarios, and any Web Services scenario that does not involve asynchronous request and response. You will, however, see INFO messages in the server log indicating that asynchronous resources have not been configured and that the asynchronous response service for Web services has not been completely deployed.

If you prefer to configure the resources manually, perform the following steps.

Table 7–2 Steps to Configure Host WebLogic Server Instance Manually for the Asynchronous Web Service

#	Step	Description
1	Invoke the Administration Console for the domain that contains the host WebLogic Server instance.	To invoke the Administration Console in your browser, enter the following URL: `http://host:port/console` where ■ *host* refers to the computer on which the Administration Server is running. ■ *port* refers to the port number where the Administration Server is listening for connection requests. The default port number for the Administration server is 7001. See "Invoking the Administration Console" in *Understanding WebLogic Web Services for Oracle WebLogic Server*.
2	Create a JMS Server.	Create a JMS Server. If a JMS server already exists, you can use it if you do not want to create a new one. See "Create JMS servers" in *Oracle WebLogic Server Administration Console Online Help*.
3	Create JMS module and define queue.	Create a JMS module, and then define a JMS queue in the module. If a JMS module already exists, you can use it if you do not want to create a new one. Target the JMS queue to the JMS server you created in the preceding step. Be sure you specify that this JMS queue is local, typically by setting the local JNDI name. See "Create JMS system modules" and "Create queues in a system module" in *Oracle WebLogic Server Administration Console Online Help*. If you want the asynchronous Web service to use the default Web services queue, set the JNDI name of the JMS queue to `weblogic.wsee.DefaultQueue`. **Clustering Considerations:** If you are using the Web service asynchronous feature in a cluster, you must: ■ Create a *local* JMS queue, rather than a distributed queue, when creating the JMS queue. ■ Explicitly target this JMS queue to each server in the cluster.
4	Create a Work Manager.	Define a Work Manager named `weblogic.wsee.mdb.DispatchPolicy`, which is used by the asynchronous request-response feature, by default. See "Create global Work Managers" in *Oracle WebLogic Server Administration Console Online Help*.
5	Tune your domain environment, as required. (Optional)	Review "Tuning Heavily Loaded Systems to Improve Web Service Performance" in *Tuning Performance of Oracle WebLogic Server*.

7.4 Writing the Asynchronous JWS File

The following example shows a simple JWS file that implements a Web service called `StockQuoteClient` that has a single method, `asyncOperation`, that in turn asynchronously invokes the `getQuote` method of the `StockQuote` service. The Java code in bold is described Section 7.4.1, "Coding Guidelines for Invoking a Web Service

Asynchronously". See Section 7.4.3, "Example of a Synchronous Invoke" to see how the asynchronous invoke differs from a synchronous invoke of the same operation.

```
package examples.webservices.async_req_res;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;
import weblogic.jws.AsyncResponse;
import weblogic.jws.AsyncFailure;

import weblogic.wsee.async.AsyncPreCallContext;
import weblogic.wsee.async.AsyncCallContextFactory;
import weblogic.wsee.async.AsyncPostCallContext;

import javax.jws.WebService;
import javax.jws.WebMethod;

import examples.webservices.async_req_res.StockQuotePortType;

import java.rmi.RemoteException;

@WebService(name="StockQuoteClientPortType",
            serviceName="StockQuoteClientService",
            targetNamespace="http://examples.org/")

@WLHttpTransport(contextPath="asyncClient",
                 serviceUri="StockQuoteClient",
                 portName="StockQuoteClientServicePort")

/**
 *  Client Web Service that invokes the StockQuote Service asynchronously.
 */

public class StockQuoteClientImpl {

  @ServiceClient(wsdlLocation="http://localhost:7001/async/StockQuote?WSDL",
                 serviceName="StockQuoteService", portName="StockQuote")

  private StockQuotePortType port;

  @WebMethod
  public void asyncOperation (String symbol, String userName)
    throws RemoteException {

    AsyncPreCallContext apc = AsyncCallContextFactory.getAsyncPreCallContext();
    apc.setProperty("userName", userName);

    try {
      port.getQuoteAsync(apc, symbol );
      System.out.println("in getQuote method of StockQuoteClient WS");

     } catch (RemoteException re) {

        System.out.println("RemoteException thrown");
        throw new RuntimeException(re);
    }

  }

  @AsyncResponse(target="port", operation="getQuote")
  public void onGetQuoteAsyncResponse(AsyncPostCallContext apc, int quote) {
```

```
    // Get the userName property we set on AsyncPreCallContext
    String userName = (String)apc.getProperty("userName");
    System.out.println("-------------------");
    System.out.println(username + " Got quote " + quote );
    System.out.println("-------------------");
  }

  @AsyncFailure(target="port", operation="getQuote")
  public void onGetQuoteAsyncFailure(AsyncPostCallContext apc, Throwable e) {
    System.out.println("-------------------");
    e.printStackTrace();
    System.out.println("-------------------");
  }

}
```

7.4.1 Coding Guidelines for Invoking a Web Service Asynchronously

The following guidelines for invoking an operation asynchronously correspond to the Java code shown in **bold** in the example described in Section 7.4, "Writing the Asynchronous JWS File". These guidelines are in addition to the standard ones for creating JWS files. See Section 7.4.3, "Example of a Synchronous Invoke" to see how the asynchronous invoke differs from a synchronous invoke of the same operation.

To invoke an operation asynchronously in your JWS file:

- Import the following WebLogic-specific JWS annotations related to the asynchronous request-response feature:

  ```
  import weblogic.jws.ServiceClient;
  import weblogic.jws.AsyncResponse;
  import weblogic.jws.AsyncFailure;
  ```

- Import the JAX-RPC stub, created later by the `jwsc` Ant task, of the port type of the Web service you want to invoke. The stub package is specified by the `packageName` attribute of the `<clientgen>` child element of `jwsc`, and the name of the stub is determined by the WSDL of the invoked Web service.

  ```
  import examples.webservices.async_req_res.StockQuotePortType;
  ```

- Import the asynchronous pre- and post-call context WebLogic APIs:

  ```
  import weblogic.wsee.async.AsyncCallContextFactory;
  import weblogic.wsee.async.AsyncPreCallContext;
  import weblogic.wsee.async.AsyncPostCallContext;
  ```

  For more information about asynchronous pre- and post-call context, see Section 7.4.2, "Using Asynchronous Pre- and Post-call Contexts". See the `"weblogic.wsee.async"` package in *Java API Reference for Oracle WebLogic Server* for additional reference information about these APIs.

- In the body of the JWS file, use the required `@ServiceClient` JWS annotation to specify the WSDL, name, and port of the Web service you will be invoking asynchronously. You specify this annotation at the field-level on a variable, whose data type is the JAX-RPC port type of the Web service you are invoking.

  ```
  @ServiceClient(
       wsdlLocation="http://localhost:7001/async/StockQuote?WSDL",
       serviceName="StockQuoteService",
       portName="StockQuote")
  ```

```
private StockQuotePortType port;
```

When you annotate a variable (in this case, `port`) with the `@ServiceClient` annotation, the Web services runtime automatically initializes and instantiates the variable, preparing it so that it can be used to invoke another Web service asynchronously.

- In the method of the JWS file which is going to invoke the `getQuote` operation asynchronously, get a pre-call asynchronous context using the context factory:

```
AsyncPreCallContext apc =
    AsyncCallContextFactory.getAsyncPreCallContext();
```

For more information about asynchronous pre- and post-call context, see Section 7.4.2, "Using Asynchronous Pre- and Post-call Contexts".

- Use the `setProperty` method of the pre-call context to create a property to store the username:

```
apc.setProperty("userName", userName);
```

- Using the stub you annotated with the `@ServiceClient` annotation, invoke the operation (in this case, `getQuote`). Instead of invoking it directly, however, invoke the asynchronous flavor of the operation, which has `Async` added on to the end of its name. The asynchronous flavor always returns `void`. Pass the asynchronous context as the first parameter:

```
port.getQuoteAsync(apc, symbol);
```

- For each operation you will be invoking asynchronously, create a method called `on`*`Operationname`*`AsyncResponse`, where *`Operationname`* refers to the name of the operation, with initial letter always capitalized. The method must return `void`, and have two parameters: the post-call asynchronous context and the return value of the operation you are invoking. Annotate the method with the `@AsyncResponse` JWS annotation; use the `target` attribute to specify the variable whose datatype is the JAX-RPC stub and the `operation` attribute to specify the name of the operation you are invoking asynchronously. Inside the body of the method, put the business logic that processes the value returned by the operation. Use the `getProperty` method of the post-call context to get the property that was set by pre-call context before invoking the asynchronous method:

```
@AsyncResponse(target="port", operation="getQuote")
public void onGetQuoteAsyncResponse(AsyncPostCallContext apc,
     int quote) {
  // Get the userName property we set on AsyncPreCallContext
  String userName = (String)apc.getProperty("userName");
  System.out.println("-------------------");
  System.out.println("Got quote " + quote );
  System.out.println("-------------------");
}
```

For more information about asynchronous pre- and post-call context, see Section 7.4.2, "Using Asynchronous Pre- and Post-call Contexts".

- For each operation you will be invoking asynchronously, create a method called `on`*`Operationname`*`AsyncFailure`, where *`Operationname`* refers to the name of the operation, with initial letter capitalized. The method must return `void`, and have two parameters: the post-call asynchronous context and a `Throwable` object, the superclass of all exceptions to handle any type of exception thrown by the

invoked operation. Annotate the method with the `@AsyncFailure` JWS annotation; use the `target` attribute to specify the variable whose datatype is the JAX-RPC stub and the `operation` attribute to specify the name of the operation you are invoking asynchronously. Inside the method, you can determine the exact nature of the exception and write appropriate Java code.

```
@AsyncFailure(target="port", operation="getQuote")
public void onGetQuoteAsyncFailure(AsyncPostCallContext apc,
    Throwable e) {
  System.out.println("-------------------");
  e.printStackTrace();
  System.out.println("-------------------");
}
```

Note: You are not required to use the `@AsyncResponse` and `@AsyncFailure` annotations, although it is a good practice because it clears up any ambiguity and makes your JWS file clean and understandable. However, in the rare use case where you want one of the `onXXX` methods to handle the asynchronous response or failure from two (or more) stubs that are invoking operations from two different Web services that have the same name, then you should explicitly NOT use these annotations. Be sure that the name of the `onXXX` methods follow the correct naming conventions exactly, as described above.

7.4.2 Using Asynchronous Pre- and Post-call Contexts

The `AsyncPreCallContext` and `AsyncPostCallContext` APIs describe asynchronous contexts that you can use in your Web service for a variety of reasons. For example:

- Set a property in the pre-context so that the method that handles the asynchronous response can distinguish between different asynchronous calls.
- Get and set contextual variables, such as the name of the user invoking the operation, their password, and so on.
- Get the name of the JAX-RPC stub that invoked a method asynchronously; and to set a time-out interval on the context.

To use asynchronous pre- and post-call contexts:

1. Import the asynchronous pre- and post-call context WebLogic APIs:

   ```
   import weblogic.wsee.async.AsyncCallContextFactory;
   import weblogic.wsee.async.AsyncPreCallContext;
   import weblogic.wsee.async.AsyncPostCallContext;
   ```

2. In the method of the JWS file that is going to invoke the asynchronous operation, get a pre-call asynchronous context using the context factory. For example:

   ```
   AsyncPreCallContext apc =
       AsyncCallContextFactory.getAsyncPreCallContext();
   ```

3. Use the pre-call context methods to operate on the asynchronous context before the asynchronous method is called. The following example uses the `setProperty` method of the pre-call context to create a property that stores the username:

   ```
   apc.setProperty("userName", userName);
   ```

4. Use the post-call context methods to operate on the asynchronous context after the asynchronous method is called. The following example uses the `getProperty` method of the post-call context to get the property that was set by pre-call context before invoking the asynchronous method:

```
String userName = (String)apc.getProperty("userName");
```

7.4.3 Example of a Synchronous Invoke

The following example shows a JWS file that invokes the `getQuote` operation of the `StockQuote` Web service synchronously. The example is shown only so you can compare it with the corresponding asynchronous invoke shown in Section 7.4, "Writing the Asynchronous JWS File".

```
package examples.webservices.async_req_res;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;

import javax.jws.WebService;
import javax.jws.WebMethod;

import java.rmi.RemoteException;

@WebService(name="SyncClientPortType",
            serviceName="SyncClientService",
            targetNamespace="http://examples.org/")

@WLHttpTransport(contextPath="syncClient",
                 serviceUri="SyncClient",
                 portName="SyncClientPort")

/**
 *  Normal service-to-service client that invokes StockQuote service
 *  synchronously.
 */

public class SyncClientImpl {

  @ServiceClient(wsdlLocation="http://localhost:7001/async/StockQuote?WSDL",
                 serviceName="StockQuoteService", portName="StockQuote")
  private StockQuotePortType port;

  @WebMethod
  public void nonAsyncOperation(String symbol) throws RemoteException {

    int quote = port.getQuote(symbol);

    System.out.println("-------------------");
    System.out.println("Got quote " + quote );
    System.out.println("-------------------");

  }

}
```

7.5 Updating the build.xml File When Using Asynchronous Request-Response

To update a `build.xml` file to generate the JWS file that invokes a Web service operation asynchronously, add `taskdefs` and a `build-clientService` target that looks something like the following; see the description after the example for details:

```
<taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JwscTask" />

<target name="build-clientService">

    <jwsc
        enableAsyncService="true"
        srcdir="src"
        destdir="${clientService-ear-dir}" >

        <jws file="examples/webservices/async_req_res/StockQuoteClientImpl.java" >

          <clientgen
            wsdl="http://${wls.hostname}:${wls.port}/async/StockQuote?WSDL"
            packageName="examples.webservices.async_req_res"/>

        </jws>

    </jwsc>

</target>
```

Use the `taskdef` Ant task to define the full classname of the `jwsc` Ant tasks.

Update the `jwsc` Ant task that compiles the client Web service to include a `<clientgen>` child element of the `<jws>` element so as to generate and compile the JAX-RPC stubs for the deployed `StockQuote` Web service. The `jwsc` Ant task automatically packages them in the generated WAR file so that the client Web service can immediately access the stubs. By default, the `jwsc` Ant task in this case generates both synchronous and asynchronous flavors of the Web service operations in the JAX-RPC stubs. You do this because the `StockQuoteClientImpl` JWS file imports and uses one of the generated classes.

7.6 Disabling The Internal Asynchronous Service

By default, every WebLogic Server instance deploys an internal asynchronous Web service that handles the asynchronous request-response feature. To specify that you do *not* want to deploy this internal service, start the WebLogic Server instance using the `-Dweblogic.wsee.skip.async.response=true` Java system property.

One reason for disabling the asynchronous service is if you use a WebLogic Server instance as a Web proxy to a WebLogic cluster. In this case, asynchronous messages will never get to the cluster, as required, because the asynchronous service on the proxy server consumes them instead. For this reason, you must disable the asynchronous service on the proxy server using the system property.

For details on specifying Java system properties to configure WebLogic Server, see "Specifying Java Options for a WebLogic Server Instance" in *Administering Server Startup and Shutdown for Oracle WebLogic Server*.

7.7 Using Asynchronous Request Response With a Proxy Server

Client applications that use the asynchronous request-response feature might not invoke the operation directly, but rather, use a proxy server. Reasons for using a proxy include the presence of a firewall or the deployment of the invoked Web service to a cluster.

In this case, the WebLogic Server instance that hosts the invoked Web service must be configured with the address and port of the proxy server. If your Web service is deployed to a cluster, you must configure every server in the cluster.

This procedure describes how to create a network channel, the primary configurable WebLogic Server resource for managing network connection. Network channels enable you to provide a consistent way to access the front-end address of a cluster. For more information about network channels, see "Understanding Network Channels" in *Administering Server Environments for Oracle WebLogic Server*.

For each server instance:

1. Create a network channel for the protocol you use to invoke the Web service. You must name the network channel `weblogic-wsee-proxy-channel-`*`XXX`*, where *`XXX`* refers to the protocol. For example, to create a network channel for HTTPS, call it `weblogic-wsee-proxy-channel-https`.

   See "Configure custom network channels" in *Oracle WebLogic Server Administration Console Online Help* for general information about creating a network channel.

2. Configure the network channel, updating the **External Listen Address** and **External Listen Port** fields with the address and port of the proxy server, respectively.

8

Using Web Services Reliable Messaging

This chapter describes Web services reliable messaging for WebLogic Java API for XML-based RPC (JAX-RPC) Web services.

This chapter includes the following topics:

Note: Web service reliable messaging requires the use of asynchronous request-response feature of WebLogic Web services. Before proceeding, you should familiarize yourself with the concepts described in Chapter 7, "Invoking a Web Service Using Asynchronous Request-Response."

8.1 Overview of Web Service Reliable Messaging

Web service reliable messaging is a framework that enables an application running on one application server to *reliably* invoke a Web service running on another application server, assuming that both servers implement the WS-ReliableMessaging specification. Reliable is defined as the ability to guarantee message delivery between the two Web Services in the presence of software component, system, or network failures.

> **Note:** Web services reliable messaging works between *any* two application servers that implement the WS-ReliableMessaging specification at http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf. In this document, however, it is assumed that the two application servers are WebLogic Server instances.
>
> Web services reliable messaging is not supported with the JMS transport feature.

WebLogic Web services conform to the WS-ReliableMessaging specification (June 2007) at http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf, which describes how two Web services running on different application servers can communicate reliably. In particular, the specification describes an interoperable protocol in which a message sent from a *source endpoint* (or client Web service) to a *destination endpoint* (or Web service whose operations can be invoked reliably) is guaranteed either to be delivered, according to one or more *delivery assurances*, or to raise an error.

A reliable WebLogic Web service provides the following delivery assurances.

Table 8–1 Delivery Assurances for Reliable Messaging

Delivery Assurance	Description
At Most Once	Messages are delivered at most once, without duplication. It is possible that some messages may not be delivered at all.
At Least Once	Every message is delivered at least once. It is possible that some messages are delivered more than once.
Exactly Once	Every message is delivered exactly once, without duplication.
In Order	Messages are delivered in the order that they were sent. This delivery assurance can be combined with one of the preceding three assurances.

> **Note:** Web services reliable messaging requires the use of asynchronous messages. Clients cannot invoke a reliable service synchronously. When invoking a reliable stub method, you *must* use the async signature—for example, xyzAsync() instead of xyz(). For clients that create SOAP messages directly, the request message is created with non-anonymous ReplyTo address.
>
> You cannot set ReplyTo to the anonymous URI. Any attempt to invoke an operation on a JAX-RPC-based reliable service (either by invoking the sync stub signature or sending a request with anonymous ReplyTo) will result in a runtime exception.

This document describes how to create the reliable and client Web services and how to configure the two WebLogic Server instances to which the Web services are deployed. See the WS-ReliableMessaging specification for detailed documentation about the architecture of Web service reliable messaging (see http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf).

8.1.1 Using WS-Policy to Specify Reliable Messaging Policy Assertions

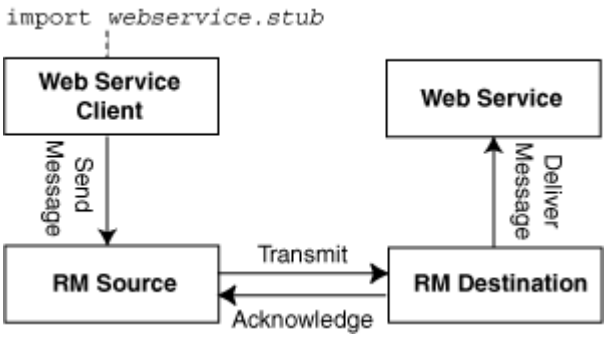
WebLogic Web services use WS-Policy files to enable a destination endpoint to describe and advertise its Web service reliable messaging capabilities and requirements. The WS-Policy files are XML files that describe features such as the version of the supported WS-ReliableMessaging specification and quality of service requirements. The WS-Policy specification (`http://www.w3.org/TR/ws-policy/`) provides a general purpose model and syntax to describe and communicate the policies of a Web service.

WebLogic Server includes pre-packaged WS-Policy files that contain typical reliable messaging assertions, as described in Appendix A, "Pre-Packaged WS-Policy Files for Reliable Messaging." If the pre-packaged WS-Policy files do not suit your needs, you must create your own WS-Policy file. See Section 8.5, "Creating the Web Service Reliable Messaging WS-Policy File" for details. See "Web Service Reliable Messaging Policy Assertion Reference" in the *WebLogic Web Services Reference for Oracle WebLogic Server* for reference information about the reliable messaging policy assertions.

8.1.2 Managing the Life Cycle of the Reliable Message Sequence

The following figure shows a one-way reliable message exchange.

Figure 8–1 Web Service Reliable Message Exchange



A *reliable message sequence* is used to track the progress of a set of messages that are exchanged reliably between an RM source and RM destination. A sequence can be used to send zero or more messages, and is identified by a string *identifier*. This identifier is used to reference the sequence when using reliable messaging.

The Web service client application sends a message for reliable delivery which is transmitted by the RM source to the RM destination. The RM destination acknowledges that the reliable message has been received and delivers it to the Web service application. The message may be retransmitted by the RM source until the acknowledgement is received.

A Web service client sends messages to a target Web service by invoking methods on a JAX-RPC *stub*. The stub is associated with the port type of the reliable Web service and represents a programmatic interface to that service. WebLogic stores the identifier for the reliable message sequence within this stub. This causes the reliable message sequence to be connected to a single JAX-RPC stub. All messages that are sent using a given stub will use the same reliable messaging sequence, regardless of the number of messages that are sent using the stub. The JAX-RPC stub is created by the `<clientgen>` child element of the "jwsc" Ant task.

Because WebLogic Server retains resources associated with the reliable sequence, it is recommended that you take steps to release these resources in a timely fashion.

WebLogic Server provides a utility class, `weblogic.wsee.reliability.WsrmUtils`, for use with the Web service reliable messaging. Use this class to perform common tasks such as set configuration options, get the sequence id, and terminate a reliable sequence.

Under normal circumstances, a reliable sequence should be retained until all messages have been sent and acknowledged by the RM destination. To facilitate the timely and proper termination of a sequence, it is recommended that you identify the final message in a reliable message sequence. Doing so indicates you are done sending messages to the RM destination and that WebLogic Server can begin looking for the final acknowledgement before automatically terminating the reliable sequence. Indicate the final message using the `weblogic.wsee.reliability.WsrmUtils.setFinalMessage()` method, passing the JAX-RPC stub being used to send messages to the RM destination.

When you identify a final message, after all messages up to and including the final message are acknowledged, the reliable message sequence is terminated, and all resources are released. Otherwise, the sequence is terminated automatically after the configured sequence expiration period is reached.

Although not recommended, you can terminate the sequence reliable message sequence regardless of whether all messages have been acknowledged using the `terminateSequence()` method. Once issued, no further reliable messages can be sent on this stub.

Note: The JAX-RPC stub is not fully initialized until shortly after the first method is invoked on the reliable Web service. When the first method is invoked, the RM source sends a `CreateSequence` message to the RM destination requesting that the RM destination create and register the reliable sequence. The RM destination, at some later time, responds with the ID for the newly created sequence. Until this response ID is received, the RM source cannot have any further communication with the RM destination and the JAX-RPC stub representing the target service at the RM destination cannot be used.

You cannot perform operations on the reliable message sequence until it is fully initialized; otherwise an error is returned. Use the `weblogic.wsee.reliability.WsrmUtils.waitForSequenceInitialization()` method to monitor whether or not the reliable message sequence has been initialized. Once the reliable sequence is initialized, this method returns the ID of the sequence.

For more information about the `WsrmUtils` utility class, see "weblogic.wsee.reliability.WsrmUtils" in *Java API Reference for Oracle WebLogic Server*.

8.2 Using Web Service Reliable Messaging: Main Steps

Configuring reliable messaging for a WebLogic Web service requires standard JMS tasks such as creating JMS servers and Store and Forward (SAF) agents, as well as Web service-specific tasks, such as adding additional JWS annotations to your JWS file. Optionally, you create WS-Policy files that describe the reliable messaging capabilities of the reliable Web service if you do not use the pre-packaged ones.

If you are using the WebLogic client APIs to invoke a reliable Web service, the client application must run on WebLogic Server. Thus, configuration tasks must be performed on both the *source* WebLogic Server instance on which the Web service that

includes client code to invoke the reliable Web service reliably is deployed, as well as the *destination* WebLogic Server instance on which the reliable Web service itself is deployed.

The following table summarizes the steps to create a reliable Web service, as well as a client Web service that invokes an operation of the reliable Web service. The procedure describes how to create the JWS files that implement the two Web services from scratch; if you want to update existing JWS files, use this procedure as a guide. The procedure also describes how to configure the source and destination WebLogic Server instances.

Table 8–2 Steps to Create and Invoke a Reliable Web Service

#	Step	Description
1	Configure the *destination* and *source* WebLogic Server instances.	You will deploy the reliable Web service to the *destination* WebLogic Server instance. For information about configuring the destination WebLogic Server instance, see Section 8.3, "Configuring the Destination WebLogic Server Instance". You will deploy the client Web service that invokes the reliable Web service to the *source* WebLogic Server instance. For information about configuring the source WebLogic Server instance, see Section 8.4, "Configuring the Source WebLogic Server Instance".
2	Create the WS-Policy file. (Optional)	Using your favorite XML or plain text editor, optionally create a WS-Policy file that describes the reliable messaging capabilities of the Web service running on the destination WebLogic Server. For details about creating your own WS-Policy file, see Section 8.5, "Creating the Web Service Reliable Messaging WS-Policy File". This step is not required if you plan to use one of the WS-Policy files that are included in WebLogic Server; see Appendix A, "Pre-Packaged WS-Policy Files for Reliable Messaging," for more information.
3	Create or update the JWS file that implements the reliable Web service.	This Web service will be deployed to the destination WebLogic Server instance. See Section 8.6, "Programming Guidelines for the Reliable JWS File".
4	Update the `build.xml` file that is used to compile the reliable Web services.	Update your `build.xml file` to include a call to the `jwsc` Ant task which will compile the reliable JWS file into a Web service. See "Running the jwsc WebLogic Web Services Ant Task" on page 3-7 for general information about using the `jwsc` task.
5	Compile and deploy the reliable JWS file.	Compile the reliable JWS file by calling the appropriate target and deploy to the destination WebLogic Server. For example: `prompt> ant build-mainService deploy-mainService`
6	Configure the reliable Web service.	Configure the reliable messaging options for the reliable Web service using the Administration Console. See Section 8.7, "Configuring Reliable Messaging for a Reliable Web Service".
7	Create or update the JWS file that implements the client Web service.	This service invokes the reliable Web service and will be deployed to the source WebLogic Server. See Section 8.8, "Programming Guidelines for the JWS File That Invokes a Reliable Web Service".
8	Update the `build.xml` file that is used to compile the client Web service.	See Section 8.9, "Updating the build.xml File for a Client of a Reliable Web Service".

Table 8–2 (Cont.) Steps to Create and Invoke a Reliable Web Service

#	Step	Description
9	Compile and deploy the client JWS file.	Compile your client JWS file by calling the appropriate target and deploy to the source WebLogic Server. For example: `prompt> ant build-clientService deploy-clientService`

Each of these steps is described in more detail in the following sections.

In addition, the following advanced topics are discussed:

- Using Reliable Messaging With MTOM—Develop a reliable Web service that uses MTOM/XOP to optimize the transmission of XML data of type `xs:base64Binary` in SOAP messages
- Client Considerations When Redeploying a Reliable Web Service—Describes client considerations for when you deploy a new version of an updated reliable WebLogic Web service alongside an older version of the same Web service.
- Using Reliable Messaging With a Proxy Server—Describes considerations when invoking a reliable Web services operations using a proxy server.

8.2.1 Prerequisites

It is assumed that you have completed the following tasks:

- You have created the *destination* and *source* WebLogic Server instances.
- You have set up an Ant-based development environment for each environment.
- You have working `build.xml` files that you can edit, for example, to add targets for running the `jwsc` Ant task and deploying the generated reliable Web service.

For more information, see "Developing JAX-RPC Web Services" on page 3-1.

8.3 Configuring the Destination WebLogic Server Instance

To configure the WebLogic Server instance on which the reliable Web service is deployed, configure the JMS and store and forward (SAF) resources.

You can configure these resources manually or you can use the Configuration Wizard to extend the WebLogic Server domain using a Web services-specific extension template. Using the Configuration Wizard greatly simplifies the required configuration steps; for details, see "Configuring Your Domain For Web Services Features" on page 3-2.

Note: Alternatively, you can use WLST to configure the resources. For information about using WLST to extend the domain, see "Configuring Existing Domains" in *Understanding the WebLogic Scripting Tool*.

A domain that does not contain Web Services resources will still boot and operate correctly for non-Web services scenarios, and any Web Services scenario that does not involve asynchronous request and response. You will, however, see INFO messages in the server log indicating that asynchronous resources have not been configured and that the asynchronous response service for Web services has not been completely deployed.

If you prefer to configure the resources manually, perform the following steps.

Table 8–3 Steps to Configure the Destination WebLogic Server Instance Manually

#	Step	Description
1	Invoke the Administration Console for the domain that contains the destination WebLogic Server.	To invoke the Administration Console in your browser, enter the following URL: `http://host:port/console` where ■ *host* refers to the computer on which the Administration Server is running. ■ *port* refers to the port number where the Administration Server is listening for connection requests. The default port number for the Administration server is 7001. See "Invoking the Administration Console" in *Understanding WebLogic Web Services for Oracle WebLogic Server*.
2	Create persistent file store. (Optional)	Optionally create a persistent store (file or JDBC) that will be used by the destination WebLogic Server to store internal Web service reliable messaging information. You can use an existing one, or the default store that always exists, if you do not want to create a new one. See "Create file stores" in *Oracle WebLogic Server Administration Console Online Help*.
3	Create a JMS Server.	Create a JMS Server. If a JMS server already exists, you can use it if you do not want to create a new one. See "Create JMS servers" in *Oracle WebLogic Server Administration Console Online Help*.

Table 8–3 (Cont.) Steps to Configure the Destination WebLogic Server Instance Manually

#	Step	Description
4	Create JMS module and define queue.	Create a JMS module, and then define a JMS queue in the module. If a JMS module already exists, you can use it if you do not want to create a new one. Target the JMS queue to the JMS server you created in the preceding step. Be sure you specify that this JMS queue is local, typically by setting the local JNDI name. Take note of the JNDI name you define for the JMS queue because you will later use it when you program the JWS file that implements your reliable Web service. See "Create JMS system modules" and "Create queues in a system module" in *Oracle WebLogic Server Administration Console Online Help*. **Clustering Considerations:** If you are using the Web service reliable messaging feature in a cluster, you must: ■ Create a *local* JMS queue, rather than a distributed queue, when creating the JMS queue. ■ Explicitly target this JMS queue to each server in the cluster.
5	Create a store and forward (SAF) agent.	You can use an existing one if you do not want to create a new one. When you create the SAF agent: ■ Set the **Agent Type** field to `Both` to enable both sending and receiving agents. ■ Be sure to target the SAF agent by clicking **Next** on the first assistant page to view the Select targets page (rather than clicking **Finish**). **Clustering Considerations:** ■ If you are using reliable messaging within a cluster, you must target the SAF agent to the cluster. See "Create Store-and-Forward agents" in *Oracle WebLogic Server Administration Console Online Help*.
6	Tune your domain environment, as required. (Optional)	Review "Tuning Heavily Loaded Systems to Improve Web service Performance" in *Tuning Performance of Oracle WebLogic Server*.
7	Restart the server.	In order for the configuration changes to take effect, you must restart the server, as described in "Starting and Stopping Servers" in *Administering Server Startup and Shutdown for Oracle WebLogic Server*.

8.4 Configuring the Source WebLogic Server Instance

Configuring the WebLogic Server instance on which the client Web service is deployed involves configuring JMS and store and forward (SAF) resources.

You can configure these resources manually or you can use the Configuration Wizard to extend the WebLogic Server domain using a Web services-specific extension template. Using the Configuration Wizard greatly simplifies the required configuration steps; for details, see "Configuring Your Domain For Web Services Features" on page 3-2.

Notes: Alternatively, you can use WLST to configure the resources. For information about using WLST to extend the domain, see "Configuring Existing Domains" in *Understanding the WebLogic Scripting Tool*.

A domain that does not contain Web Services resources will still boot and operate correctly for non-Web services scenarios, and any Web Services scenario that does not involve asynchronous request and response. You will, however, see INFO messages in the server log indicating that asynchronous resources have not been configured and that the asynchronous response service for Web services has not been completely deployed.

If you prefer to configure the resources manually, perform the following steps.

Table 8–4 Steps to Configure the Source WebLogic Server Instance

#	Step	Description
1	Invoke the Administration Console for the domain that contains the source WebLogic Server.	To invoke the Administration Console in your browser, enter the following URL: `http://host:port/console` where ■ *host* refers to the computer on which the Administration Server is running. ■ *port* refers to the port number where the Administration Server is listening for connection requests. The default port number for the Administration server is 7001. See "Invoking the Administration Console" in *Understanding WebLogic Web Services for Oracle WebLogic Server*.
2	Create persistent file store. (Optional)	Optionally create a persistent store (file or JDBC) that will be used by the source WebLogic Server to store internal Web service reliable messaging information. You can use an existing one, or the default store that always exists, if you do not want to create a new one. See "Create file stores" in *Oracle WebLogic Server Administration Console Online Help*.
3	Create as store and forward (SAF) agent.	You can use an existing one if you do not want to create a new one. When you create the SAF agent, set the **Agent Type** field to `Both` to enable both sending and receiving agents. See "Create Store-and-Forward agents" in *Oracle WebLogic Server Administration Console Online Help*.
6	Tune your domain environment, as required. (Optional)	Review "Tuning Heavily Loaded Systems to Improve Web service Performance" in *Tuning Performance of Oracle WebLogic Server*.
7	Restart the server.	In order for the configuration changes to take effect, you must restart the server, as described in "Starting and Stopping Servers" in *Administering Server Startup and Shutdown for Oracle WebLogic Server*.

8.5 Creating the Web Service Reliable Messaging WS-Policy File

A WS-Policy file is an XML file that contains policy assertions that comply with the WS-Policy specification. In this case, the WS-Policy file contains Web service reliable messaging policy assertions.

WebLogic Server includes pre-packaged WS-Policy files that contain typical reliable messaging assertions that you can use if you do not want to create your own WS-Policy file. The pre-packaged WS-Policy files are listed in the following table.

> **Note:** The `DefaultReliability.xml` and `LongRunningReliability.xml` files are deprecated in this release. Use of the `DefaultReliability1.1.xml`, `Reliability1.1_SequenceTransportSecurity`, or `Reliability1.0_1.1.xml` file is recommended and required to comply with the 1.1 version of the WS-ReliableMessaging specification at `http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf`.

Table 8–5 Pre-packaged WS-Policy Files

Pre-packaged WS-Policy File	Description
`DefaultReliability1.1.xml`	Specifies policy assertions related to quality of service. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.pdf`. See Section A.1, "DefaultReliability1.1.xml (WS-Policy File)".
`Reliability1.1_SequenceTransportSecurity`	Specifies policy assertions related to transport-level security and quality of service. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.pdf`. See Section A.2, "Reliability1.1_SequenceTransportSecurity.xml (WS-Policy File)".
`Reliability1.0_1.1.xml`	Combines 1.1 and 1.0 WS Reliable Messaging policy assertions. This sample relies on smart policy selection to determine the policy assertion that is applied at runtime. See Section A.4, "Reliability1.0_1.1.xml (WS-Policy.xml File)"
`DefaultRelibility.xml`	**Deprecated.** The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion Version 1.0 at `http://schemas.xmlsoap.org/ws/2005/02/rm/WS-RMPolicy.pdf`. In this release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration. Specifies typical values for the reliable messaging policy assertions, such as inactivity timeout of 10 minutes, acknowledgement interval of 200 milliseconds, and base retransmission interval of 3 seconds. See Section A.5, "DefaultReliability.xml (WS-Policy File) [Deprecated]".
`LongRunningReliability.xml`	**Deprecated.** The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion Version 1.0 for long running processes. In this release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration. Similar to the preceding default reliable messaging WS-Policy file, except that it specifies a much longer activity timeout interval (24 hours.) See Section A.6, "LongRunningReliability.xml (WS-Policy File) [Deprecated]".

You can use one of the pre-packaged reliable messaging WS-Policy files included in WebLogic Server; these files are adequate for most use cases. You cannot modify the pre-packaged files. If the values do not suit your needs, you must create a custom WS-Policy file. The following sections describe how to create a custom WS-Policy file.

- Section 8.5.1, "Creating a Custom WS-Policy File Using WS-ReliableMessaging Policy Assertions Version 1.1"

8.5.1 Creating a Custom WS-Policy File Using WS-ReliableMessaging Policy Assertions Version 1.1

This section describes how to create a custom WS-Policy file that contains Web service reliable messaging assertions that are based on WS Reliable Messaging Policy Assertion Version 1.1 at http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.pdf. In the current release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration.

The root element of the WS-Policy file is `<Policy>` and it should include the following namespace declaration:

```
<wsp:Policy
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
```

You wrap all Web service reliable messaging policy assertions inside of a `<wsrmp:RMAssertion>` element. This element should include the following namespace declaration for using Web service reliable messaging policy assertions:

```
<wsrmp:RMAssertion
   xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
```

The following table lists the Web service reliable messaging assertions that you can specify in the WS-Policy file. The order in which the assertions appear is important. You can specify the following assertions; the order they appear in the following list is the order in which they should appear in your WS-Policy file:

Table 8–6 Web Service Reliable Messaging Assertions (Version 1.1)

Assertion	Description
`<wsrmp:SequenceSTR>`	To secure messages in a reliable sequence, the runtime will use the `wsse:SecurityTokenReference` that is referenced in the `CreateSequence` message. You can only specify one security assertion; that is, you can specify `wsrmp:SequenceSTR` or `wsrmp:SequenceTransportSecurity`, but not both.
`<wsrmp:SequenceTransportSecurity>`	To secure messages in a reliable sequence, the runtime will use the SSL transport session that is used to send the `CreateSequence` message. This assertion must be used in conjunction with the `sp:TransportBinding` assertion that requires the use of some transport-level security mechanism (for example, `sp:HttpsToken`). You can only specify one security assertion; that is, you can specify `wsrmp:SequenceSTR` or `wsrmp:SequenceTransportSecurity`, but not both.
`<wsrm:DeliveryAssurance>`	Delivery assurance (or quality of service) of the Web service. Valid values are `AtMostOnce`, `AtLeastOnce`, `ExactlyOnce`, and `InOrder`. You can set one of the delivery assurances defined in the following table. If not set, the delivery assurance defaults to `ExactlyOnce`.

The following example shows a simple Web service reliable messaging WS-Policy file:

```
<?xml version="1.0"?>

<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
```

```
    <wsrmp:RMAssertion
         xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
    <wsrmp:SequenceTransportSecurity/>
    <wsrmp:DeliveryAssurance>
      <wsp:Policy>
        <wsrmp:ExactlyOnce/>
      </wsp:Policy>
    </wsrmp:DeliveryAssurance>
  </wsrmp:RMAssertion>
</wsp:Policy>
```

For more information about Reliable Messaging policy assertions in the WS-Policy file, see "Web Service Reliable Messaging Policy Assertion Reference" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

8.5.2 Creating a Custom WS-Policy File Using WS-ReliableMessaging Policy Assertions Version 1.0 (Deprecated)

This section describes how to create a custom WS-Policy file that contains Web service reliable messaging assertions that are based on WS Reliable Messaging Policy Assertion Version 1.0 at http://schemas.xmlsoap.org/ws/2005/02/rm/WS-RMPolicy.pdf.

> **Note:** In the current release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration.

The root element of the WS-Policy file is `<Policy>` and it should include the following namespace declarations for using Web service reliable messaging policy assertions:

```
<wsp:Policy
   xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
   xmlns:beapolicy="http://www.bea.com/wsrm/policy">
```

You wrap all Web service reliable messaging policy assertions inside of a `<wsrm:RMAssertion>` element. The assertions that use the `wsrm:` namespace are standard ones defined by the WS-ReliableMessaging specification at http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf. The assertions that use the `beapolicy:` namespace are WebLogic-specific. See "Web Service Reliable Messaging Policy Assertion Reference" in the *WebLogic Web Services Reference for Oracle WebLogic Server* for details.

The following table lists the Web service reliable messaging assertions that you can specify in the WS-Policy file. All Web service reliable messaging assertions are optional, so only set those whose default values are not adequate. The order in which the assertions appear is important. You can specify the following assertions; the order they appear in the following list is the order in which they should appear in your WS-Policy file,

Table 8–7 Web Service Reliable Messaging Assertions (Version 1.0)

Assertion	Description
<wsrm:InactivityTimeout>	Number of milliseconds, specified with the Milliseconds attribute, which defines an inactivity interval. After this amount of time, if the destination endpoint has not received a message from the source endpoint, the destination endpoint may consider the sequence to have terminated due to inactivity. The same is true for the source endpoint. By default, sequences never timeout.
<wsrm:BaseRetransmissionInterval>	Interval, in milliseconds, that the source endpoint waits after transmitting a message and before it retransmits the message if it receives no acknowledgment for that message. Default value is set by the SAF agent on the source endpoint's WebLogic Server instance.
<wsrm:ExponentialBackoff>	Specifies that the retransmission interval will be adjusted using the exponential backoff algorithm. This element has no attributes.
<wsrm:AcknowledgmentInterval>	Maximum interval, in milliseconds, in which the destination endpoint must transmit a stand-alone acknowledgement. The default value is set by the SAF agent on the destination endpoint's WebLogic Server instance.
<beapolicy:Expires>	Amount of time after which the reliable Web service expires and does not accept any new sequence messages. The default value is to never expire. This element has a single attribute, Expires, whose data type is an XML Schema duration type (see http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#duration). For example, if you want to set the expiration time to one day, use the following: <beapolicy:Expires Expires="P1D" />.
<beapolicy:QOS>	Delivery assurance level, as described in Table 8–9. The element has one attribute, QOS, which you set to one of the following values: AtMostOnce, AtLeastOnce, or ExactlyOnce. You can also include the InOrder string to specify that the messages be in order. The default value is ExactlyOnce InOrder. This element is typically not set.

The following example shows a simple Web service reliable messaging WS-Policy file:

```
<?xml version="1.0"?>

<wsp:Policy
   xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
   xmlns:beapolicy="http://www.bea.com/wsrm/policy"
  >

 <wsrm:RMAssertion>

   <wsrm:InactivityTimeout
      Milliseconds="600000" />
   <wsrm:BaseRetransmissionInterval
      Milliseconds="500" />
   <wsrm:ExponentialBackoff />
   <wsrm:AcknowledgementInterval
      Milliseconds="2000" />

 </wsrm:RMAssertion>

</wsp:Policy>
```

For more information about Reliable Messaging policy assertions in the WS-Policy file, see "Web Service Reliable Messaging Policy Assertion Reference" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

8.5.3 Using Multiple Policy Alternatives

You can configure multiple policy alternatives—also referred to as *smart policy alternatives*—for a single Web service by creating a custom policy file. At runtime, WebLogic Server selects which of the configured policies to apply. It excludes policies that are not supported or have conflicting assertions and selects the appropriate policy, based on your configured preferences, to verify incoming messages and build the response messages.

The following example provides an example of a security policy that supports both 1.1and 1.0 WS-Reliable Messaging. Each policy alternative is enclosed in a `<wsp:All>` element.

> **Note:** The 1.0 Web service reliable messaging assertions are prefixed by `wsrmp10`.

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <wsrmp10:RMAssertion
       xmlns:wsrmp10="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
        <wsrmp10:InactivityTimeout Milliseconds="1200000"/>
        <wsrmp10:BaseRetransmissionInterval Milliseconds="60000"/>
        <wsrmp10:ExponentialBackoff/>
        <wsrmp10:AcknowledgementInterval Milliseconds="800"/>
      </wsrmp10:RMAssertion>
    </wsp:All>
    <wsp:All>
      <wsrmp:RMAssertion
           xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
        <wsrmp:SequenceSTR/>
        <wsrmp:DeliveryAssurance>
          <wsp:Policy>
            <wsrmp:AtMostOnce/>
          </wsp:Policy>
        </wsrmp:DeliveryAssurance>
      </wsrmp:RMAssertion>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

For more information about multiple policy alternatives, see "Smart Policy Selection" in "Configuring Message-Level Security" in *Securing WebLogic Web Services for Oracle WebLogic Server*.

8.6 Programming Guidelines for the Reliable JWS File

This section describes how to create the JWS file that implements the reliable Web service.

The following JWS annotations are used in the JWS file that implements a reliable Web service.

Table 8–8 JWS Annotations for Reliable Messaging

Annotation	Description
`@weblogic.jws.Policy`	Required. Specifies that the Web service has a WS-Policy file attached to it that contains reliable messaging assertions. See Section 8.6.1, "Using the @Policy Annotation".
`@javax.jws.Oneway`	Required only if you invoke the reliable Web service operation synchronously (that is, you are not using the asynchronous request-response feature). See Section 8.6.2, "Using the @Oneway Annotation".
`@weblogic.jws.BufferQueue`	Optional. Specifies the JNDI name of the JMS queue which WebLogic Server uses to store reliable messages internally. See Section 8.6.3, "Using the @BufferQueue Annotation".
`@weblogic.jws.ReliabilityBuffer`	Optional. Specifies the number of times WebLogic Server should attempt to deliver the message from the JMS queue to the Web service implementation and the amount of time that the server should wait in between retries. See Section 8.6.4, "Using the @ReliabilityBuffer Annotation"

The following example shows a simple JWS file that implements a reliable Web service; see the explanation after the example for coding guidelines that correspond to the Java code in **bold**.

```
package examples.webservices.reliable;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.Oneway;

import weblogic.jws.WLHttpTransport;

import weblogic.jws.ReliabilityBuffer;
import weblogic.jws.BufferQueue;
import weblogic.jws.Policy;

/**
 * Simple reliable Web Service.
 */

@WebService(name="ReliableHelloWorldPortType",
             serviceName="ReliableHelloWorldService")

@WLHttpTransport(contextPath="ReliableHelloWorld",
      serviceUri="ReliableHelloWorld",
      portName="ReliableHelloWorldServicePort")

@Policy(uri="ReliableHelloWorldPolicy.xml",
          direction=Policy.Direction.both,
          attachToWsdl=true)
@BufferQueue(name="webservices.reliable.queue")

public class ReliableHelloWorldImpl {

  private static String onewaySavedInput = null;

/**
 * A one-way helloWorld method that saves the given string for later
 * concatenation to the end of the message passed into helloWorldReturn.
```

```
 */
  @WebMethod()
  @Oneway()
  @ReliabilityBuffer(retryCount=10, retryDelay="10 seconds")

  public void helloWorld(String input) {
    System.out.println(" Hello World " + input);
    onewaySavedInput = input;
   }

/**
 * This echo method concatenates the saved message from helloWorld
 * onto the end of the provided message, and returns it.
 */
  @WebMethod()
  @ReliabilityBuffer(retryCount=10, retryDelay="10 seconds")

  public String echo(String input2) {
    System.out.println(" Hello World " + input2 + onewaySavedInput);
    return input + onewaySavedInput;
  }
}
```

In the example, the custom `ReliableHelloWorldPolicy.xml` policy file is attached to the Web service at the class level, which means that the policy file is applied to all public operations of the Web service. The policy file is applied only to the request Web service message (as required by the reliable messaging feature) and it is attached to the WSDL file. For information about the pre-packaged policies available and creating a custom policy, see Section 8.5, "Creating the Web Service Reliable Messaging WS-Policy File".

The JMS queue that WebLogic Server uses internally to enable the Web service reliable messaging has a JNDI name of `webservices.reliable.queue`, as specified by the `@BufferQueue` annotation.

The `helloWorld()` method has been marked with both the `@WebMethod` and `@Oneway` JWS annotations, which means it is a public operation called `helloWorld`. Because of the `@Policy` annotation, the operation can be invoked reliably. The Web services runtime attempts to deliver reliable messages to the service a maximum of 10 times, at 10-second intervals, as described by the `@ReliabilityBuffer` annotation. The message may require re-delivery if, for example, the transaction is rolled back or otherwise does not commit.

The `echo()` method has been marked with the `@WebMethod` and JWS annotation, which means it is a public operation called `echo`. Because of the `@Policy` annotation, the operation can be invoked reliably. It uses the same reliability buffer configuration as the `helloWorld()` method.

8.6.1 Using the @Policy Annotation

Use the `@Policy` annotation in your JWS file to specify that the Web service has a WS-Policy file attached to it that contains reliable messaging assertions. WebLogic Server delivers a set of pre-packaged WS-Policy files, as described in Appendix A, "Pre-Packaged WS-Policy Files for Reliable Messaging."

Follow the following guidelines when using the `@Policy` annotation for Web service reliable messaging:

- Use the `uri` attribute to specify the build-time location of the policy file, as follows:

- If you have created your own WS-Policy file, specify its location relative to the JWS file. For example:

```
@Policy(uri="ReliableHelloWorldPolicy.xml",
          direction=Policy.Direction.both,
          attachToWsdl=true)
```

  In this example, the `ReliableHelloWorldPolicy.xml` file is located in the same directory as the JWS file.

- To specify one of the pre-packaged WS-Policy files or a WS-Policy file that is packaged in a shared Java EE library, use the `policy:` prefix along with the name and path of the policy file. This syntax tells the `jwsc` Ant task at build-time *not* to look for an actual file on the file system, but rather, that the Web service will retrieve the WS-Policy file from WebLogic Server at the time the service is deployed.

  Note: Shared Java EE libraries are useful when you want to share a WS-Policy file with multiple Web services that are packaged in different Enterprise applications. As long as the WS-Policy file is located in the `META-INF/policies` or `WEB-INF/policies` directory of the shared Java EE library, you can specify the policy file in the same way as if it were packaged in the same archive at the Web service. See "Creating Shared Java EE Libraries and Optional Packages" in *Developing Applications for Oracle WebLogic Server* for information about creating libraries and setting up your environment so the Web service can locate the policy files.

- To specify that the policy file is published on the Web, use the `http:` prefix along with the URL, as shown in the following example:

```
@Policy(uri="http://someSite.com/policies/mypolicy.xml"
        direction=Policy.Direction.both,
        attachToWsdl=true)
```

- By default, WS-Policy files are applied to both the request (inbound) and response (outbound) SOAP messages. You can change this default behavior with the `direction` attribute by setting the attribute to `Policy.Direction.inbound` or `Policy.Direction.outbound`.
- You can specify whether the Web service requires the operations to be invoked reliably and have the responses delivered reliably using the `wsp:optional` attribute within the policy file specified by `uri`.

  If the `optional` attribute is set to `false` for outbound on any operation, then:

  - The client must provide an *offer sequence* (`<wsrm: Offer...>` as described in the WS-ReliableMessaging specification at `http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf`) for use when sending reliable responses.
  - Responses will be sent reliably for all operations requiring a response.

  If the `optional` attribute is set to `true` for outbound on all operations, then:

  - The client is not required to provide an offer sequence.
  - Responses will be sent reliably if the client provides an offer sequence; otherwise, responses will be sent non-reliably.

- Set the `attachToWsdl` attribute of the `@Policy` annotation to specify whether the policy file should be attached to the WSDL file that describes the public contract of the Web service. Typically, you want to publicly publish the policy so that client applications know the reliable messaging capabilities of the Web service. For this reason, the default value of this attribute is `true`.

For more information about the `@Policy` annotation, see "weblogic.jws.Policy" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

8.6.2 Using the @Oneway Annotation

If you plan on invoking the reliable Web service operation synchronously (or in other words, *not* using the asynchronous request-response feature), then you must annotate the implementing method with the `@Oneway` annotation to specify that the method is one-way. This means that the method cannot return a value, but rather, must explicitly return `void`.

Conversely, if the method is *not* annotated with the `@Oneway` annotation, then you must invoke it using the asynchronous request-response feature. If you are unsure how the operation is going to be invoked, consider creating two flavors of the operation: synchronous and asynchronous.

See Chapter 7, "Invoking a Web Service Using Asynchronous Request-Response," and Chapter 11, "Using the Asynchronous Features Together."

8.6.3 Using the @BufferQueue Annotation

Use the `@BufferQueue` annotation to specify the JNDI name of the JMS queue which WebLogic Server uses to store reliable messages internally. The JNDI name is the one you configured when creating a JMS queue in step 4 in Section 8.3, "Configuring the Destination WebLogic Server Instance".

The `@BufferQueue` annotation is optional; if you do not specify it in your JWS file then WebLogic Server uses a queue with a JNDI name of `weblogic.wsee.DefaultQueue`. You must, however, still explicitly create a JMS queue with this JNDI name using the Administration Console.

For more information about the `@BufferQueue` annotation, see "weblogic.jws.BufferQueue" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

8.6.4 Using the @ReliabilityBuffer Annotation

Use the `@ReliabilityBuffer` annotation to specify the number of times WebLogic Server should attempt to deliver the message from the JMS queue to the Web service implementation and the amount of time that the server should wait in between retries.

Use the `retryCount` attribute to specify the number of retries and the `retryDelay` attribute to specify the wait time. The format of the `retryDelay` attribute is a number and then one of the following strings:

- `seconds`
- `minutes`
- `hours`
- `days`
- `years`

For example, to specify a retry count of 20 and a retry delay of two days, use the following syntax:

```
@ReliabilityBuffer(retryCount=20, retryDelay="2 days")
```

The retry count and delay default to 3 and 5 seconds, respectively.

For more information about the `@ReliabilityBuffer` annotation, see "weblogic.jws.ReliabilityBuffer" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

8.7 Configuring Reliable Messaging for a Reliable Web Service

If necessary, you can edit the reliable message configuration options for a reliable Web service that are stored in the `weblogic-webservices.xml` descriptor by updating the application *deployment plan*. The deployment plan associates new values with specific locations in the descriptors for your application. At deployment time, a deployment plan is merged with the descriptors in the application by applying the values in its variable assignments to the locations in the application descriptors to which the variables are linked.

The following table summarizes the reliable messaging options that can be configured for the reliable Web service.

Table 8–9 Configuration Options for Reliable Messaging

Configuration Option	Description
Customize Reliable Message Configuration	Flag that specifies whether you want to customize the reliable message configuration defined in the Web service descriptor or deployment plan at the Web service endpoint level. This flag is available only when configuring reliable messaging at the Web service endpoint level If not checked, the reliable message configuration specified for the WebLogic Server is used. **Note:** This flag does not reflect the configuration of Reliable Messaging in other forms, such as, WS-RM policy directly specified in the WSDL.
Base Retransmission Interval	Interval of time that must pass before a message is retransmitted to the RM destination. If the source endpoint does not receive an acknowledgement for a given message within the specified interval, the source endpoint retransmits the message. The source endpoint can modify this retransmission interval at any point during the lifetime of the sequence of messages. This element can be used in conjunction with the Retransmission Exponential Backoff element to specify the algorithm that is used to adjust the retransmission interval. The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`, where `n`Y specifies the number of years, `n`M specifies the number of months, `n`D specifies the number of days, `T` is the date/time separator, `n`H specifies the number of hours, `n`M specifies the number of minutes, and `n`S specifies the number of seconds. This value defaults to `P0DT3S` (3 seconds).

Table 8–9 (Cont.) Configuration Options for Reliable Messaging

Configuration Option	Description
Enable Retransmission Exponential Backoff	Flag that specifies whether the message retransmission interval will be adjusted using the exponential backoff algorithm. This element is used in conjunction with the Base Retransmission Interval element. If a destination endpoint does not acknowledge a sequence of messages for the time interval specified by the Base Retransmission Interval, the exponential backoff algorithm is used for timing successive retransmissions by the source endpoint, should the message continue to go unacknowledged. The exponential backoff algorithm specifies that successive retransmission intervals should increase exponentially, based on the base retransmission interval. For example, if the base retransmission interval is 2 seconds, and the exponential backoff element is set, successive retransmission intervals if messages continue to go unacknowledged are 2, 4, 8, 16, 32, and so on. This value defaults to false, the same retransmission interval is used in successive retries; the interval does not increase exponentially.
Acknowledgement Interval	Maximum interval during which the destination endpoint must transmit a stand-alone acknowledgement. A destination endpoint can send an acknowledgement on the return message immediately after it has received a message from a source endpoint, or it can send one separately as a stand-alone acknowledgement. If a return message is not available to send an acknowledgement, a destination endpoint may wait for up to the acknowledgement interval before sending a stand-alone acknowledgement. If there are no unacknowledged messages, the destination endpoint may choose not to send an acknowledgement. The value specified must be a positive value and conform to the XML schema duration lexical format, `P`*n*`Y`*n*`M`*n*`DT`*n*`H`*n*`M`*n*`S`, where *n*`Y` specifies the number of years, *n*`M` specifies the number of months, *n*`D` specifies the number of days, `T` is the date/time separator, *n*`H` specifies the number of hours, *n*`M` specifies the number of minutes, and *n*`S` specifies the number of seconds. This value defaults to `P0DT0.2S` (200 milliseconds).
Inactivity Timeout	Inactivity interval. If, during the inactivity timeout interval, an endpoint (the RM source or destination) has not received messages application or control messages, the endpoint may consider the RM sequence to have been terminated due to inactivity. The value specified must be a positive value and conform to the XML schema duration lexical format, `P`*n*`Y`*n*`M`*n*`DT`*n*`H`*n*`M`*n*`S`, where *n*`Y` specifies the number of years, *n*`M` specifies the number of months, *n*`D` specifies the number of days, `T` is the date/time separator, *n*`H` specifies the number of hours, *n*`M` specifies the number of minutes, and *n*`S` specifies the number of seconds. This value defaults to `P0DT600S` (600 seconds).

Table 8–9 (Cont.) Configuration Options for Reliable Messaging

Configuration Option	Description
Sequence Expiration	Expiration time for a sequence regardless of activity. The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`, where `n`Y specifies the number of years, `n`M specifies the number of months, `n`D specifies the number of days, `T` is the date/time separator, `n`H specifies the number of hours, `n`M specifies the number of minutes, and `n`S specifies the number of seconds. This value defaults to `P1D` (1 day).
Buffer Retry Count	Number of times to retry a reliable request. This value defaults to 3.
Buffer Retry Delay	Amount of time to wait before retrying a reliable request. The retry attempts are between the client's request message on the JMS queue and delivery of the message to the Web service implementation. The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`, where `n`Y specifies the number of years, `n`M specifies the number of months, `n`D specifies the number of days, `T` is the date/time separator, `n`H specifies the number of hours, `n`M specifies the number of minutes, and `n`S specifies the number of seconds. This value defaults to `P0DT5S` (5 seconds).

You can set the reliable messaging configuration options using the Administration Console or WLST, as described in the following sections.

- Section 8.7.1, "Using the Administration Console"
- Section 8.7.2, "Using WLST"

8.7.1 Using the Administration Console

To configure reliable messaging for the Web service endpoint using the Administration Console:

1. Invoke the Administration Console, as described in "Invoking the Administration Console" in *Understanding WebLogic Web Services for Oracle WebLogic Server*.
2. In the left navigation pane, select **Deployments**.
3. Click the name of the Web service in the Deployments table.
4. Select the **Configuration** tab, then the **Ports** tab.
5. Click the name of the Web service endpoint in the Ports table.
6. Select the **Reliable Message** tab.
7. Click **Customize Reliable Message Configuration** and follow the instructions to save the deployment plan, if required.
8. Set the reliable messaging properties, as required.
9. Click **Save**.

8.7.2 Using WLST

For a complete description and example of using WLST to update an application's deployment plan to configure reliable messaging, see "Updating the Deployment Plan" in *Understanding the WebLogic Scripting Tool*.

For your reference, the following table summarizes the XPath values for the WS-RM configuration options.

Table 8–10 WS_RM Configuration Variable Names and XPath Values

Configuration Option	Example Variable Name	XPath Value
Inactivity Timeout	`ReliabilityConfig_ InactivityTimeout`	`/weblogic-webservices/webservice-description/ [webservice-description-name="`*`service_name`*`"]/port-component/ [port-component-name="`*`port_name`*`"] /reliability-config/inactivity-timeout`
Base Retransmission Interval	`ReliabilityConfig_ BaseRetransmission Interval`	`/weblogic-webservices/webservice-description/ [webservice-description-name= "`*`service_name`*`"]/port-component/ [port-component-name="`*`port-name`*`"] /reliability-config/base-retransmission-interval`
Retransmission Exponential Backoff	`ReliabilityConfig_ Retransmission ExponentialBackoff`	`/weblogic-webservices/webservice-description/ [webservice-description-name= "`*`service_name`*`"]/port-component/ [port-component-name="`*`port-name`*`"]/reliability- config/retransmission-exponential-backoff`
Acknowledgement Interval	`ReliabilityConfig_ AcknowledgementInterval`	`/weblogic-webservices/webservice-description/ [webservice-description-name= "`*`service_name`*`"]/port-component/ [port-component-name="`*`port-name`*`"] /reliability-config/acknowledgement-interval`
Sequence Expiration	`ReliabilityConfig_ SequenceExpiration`	`/weblogic-webservices/webservice-description/ [webservice-description-name= "`*`service_name`*`"]/port-component/ [port-component-name="`*`port-name`*`"] /reliability-config/sequence-expiration`
Buffer Retry Count	`ReliabilityConfig_ BufferRetryCount`	`/weblogic-webservices/webservice-description/ [webservice-description-name= "`*`service_name`*`"]/port-component/ [port-component-name="`*`port-name`*`"] /reliability-config/buffer-retry-count`
Buffer Retry Delay	`ReliabilityConfig_ BufferRetryDelay`	`/weblogic-webservices/webservice-description/ [webservice-description-name= "`*`service_name`*`"]/port-component/ [port-component-name="`*`port-name`*`"] /reliability-config/buffer-retry-delay`

8.8 Programming Guidelines for the JWS File That Invokes a Reliable Web Service

If you are using the WebLogic client APIs, you must invoke a reliable Web service from within a Web service; you cannot invoke a reliable Web service from a stand-alone client application.

The following example shows a simple JWS file for a Web service that invokes a reliable operation from the service described in Section 8.6, "Programming Guidelines for the Reliable JWS File".

```
package examples.webservices.reliable;

import java.rmi.RemoteException;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.xml.rpc.Stub;
import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;
import weblogic.jws.ReliabilityErrorHandler;

import weblogic.jws.AsyncFailure;
import weblogic.jws.AsyncResponse;

import examples.webservices.reliable.ReliableHelloWorldPortType;

import weblogic.wsee.reliability.ReliabilityErrorContext;
import weblogic.wsee.reliability.ReliableDeliveryException;
import weblogic.wsee.reliability.WsrmUtils;

@WebService(name="ReliableClientPortType",
            serviceName="ReliableClientService")

@WLHttpTransport(contextPath="ReliableClient",
                 serviceUri="ReliableClient",
                 portName="ReliableClientServicePort")

public class ReliableClientImpl
{
  private static String responseMessage = null;

  @ServiceClient(
     serviceName="ReliableHelloWorldService",
     portName="ReliableHelloWorldServicePort")

  private ReliableHelloWorldPortType port;

  @WebMethod
  public void callHelloWorld(String input, String input2, String serviceUrl)
     throws RemoteException {

    ((Stub)port)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY, serviceUrl);

    port.helloWorld(input);

    System.out.println(" Invoked the ReliableHelloWorld.helloWorld operation
reliably." );
    WsrmUtils.setFinalMessage((Stub)port);
    port.echo(input2);
    System.out.println(" Invoked the ReliableHelloWorld.echo operation reliably."
);
  }

  @AsyncResponse(target = "port", operation = "echo")
  public void onEchoAsyncResponse(String msg) {
```

```
    System.out.println("ClientService: Got async response for request : " + msg);
    responseMessage = msg;
  }

  @AsyncFailure(target = "port", operation = "echo")
  public void onEchoAsyncFailure(Throwable t) {
    System.out.println("ClientService: Got async FAILURE for request : " + t);
    t.printStackTrace();
  }

  @ReliabilityErrorHandler(target="port")
  public void onReliableMessageDeliveryError(ReliabilityErrorContext ctx) {

    ReliableDeliveryException fault = ctx.getFault();
    String message = null;
    if (fault != null) {
      message = ctx.getFault().getMessage();
    }
    String operation = ctx.getOperationName();
    System.out.println("Reliable operation " + operation + " may have not invoked.
The error message is " + message);
  }

}
```

As illustrated in the previous examples (in **bold** text), follow these guidelines when programming the JWS file that invokes a reliable Web service:

- Import the `@ServiceClient` and `@ReliabilityErrorHandler` JWS annotations:

  ```
  import weblogic.jws.ServiceClient;
  import weblogic.jws.ReliabilityErrorHandler;
  ```

- Import the WebLogic APIs that you will use in the method that handles the error that results when the client Web service does not receive an acknowledgement of message receipt from the reliable Web service:

  ```
  import weblogic.wsee.reliability.ReliabilityErrorContext;
  import weblogic.wsee.reliability.ReliableDeliveryException;
  ```

- Import the APIs used for asynchronous response and failure.

  ```
  import weblogic.jws.AsyncFailure;
  import weblogic.jws.AsyncResponse;
  ```

- Import the JAX-RPC stub, created later by the `<clientgen>` child element of the `jwsc` Ant task, of the port type of the reliable Web service you want to invoke. The stub package is specified by the `packageName` attribute of `<clientgen>`, and the name of the stub is determined by the WSDL of the invoked Web service.

  ```
  import examples.webservices.reliable.ReliableHelloWorldPortType;
  ```

- Import the APIs used for life cycle management (to set properties and specify the final message later).

  ```
  import javax.xml.rpc.Stub;
  import weblogic.wsee.reliability.WsrmUtils;
  ```

- In the body of the JWS file, use the `@ServiceClient` JWS annotation to specify the name and port of the reliable Web service you want to invoke. You specify this

annotation at the field-level on a private variable, whose data type is the JAX-RPC port type of the Web service you are invoking.

```
@ServiceClient(
     serviceName="ReliableHelloWorldService",
     portName="ReliableHelloWorldServicePort")

  private ReliableHelloWorldPortType port;
```

- Use the `port._setProperty` method to dynamically specify the target service endpoint address within the Web service client.

```
((Stub)port)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY, serviceUrl);
```

- Using the stub you annotated with the `@ServiceClient` annotation, invoke the `helloWorld` reliable operation:

```
port.helloWorld(input);
```

  Because the operation has been marked one-way, it does not return a value.

- Create a method that handles the error when the client Web service does not receive an acknowledgement from the reliable Web service that the latter has received a message and annotate this method with the `@weblogic.jws.ReliabilityErrorHandler` annotation:

```
@ReliabilityErrorHandler(target="port")
public void onReliableMessageDeliveryError(ReliabilityErrorContext ctx) {
    ReliableDeliveryException fault = ctx.getFault();
    String message = null;
    if (fault != null) {
      message = ctx.getFault().getMessage();
    }
    String operation = ctx.getOperationName();
    System.out.println("Reliable operation " + operation + " may have not
invoked. The error message is " + message);
  }
```

  This method takes `ReliabilityErrorContext` as its single parameter and returns `void`.

  See "weblogic.jws.ReliabilityErrorHandler" in *WebLogic Web Services Reference for Oracle WebLogic Server* for details about programming this error-handling method.

- Because the service is not conversational, any state kept in the port field will be lost when this method returns. In the case of reliable messaging, this state includes the ID of the reliable sequence being used to send messages. The `setFinalMessage` method specifies that this is the final message to be sent on this sequence. This will allow the reliable messaging subsystem to proactively clean up the reliable sequence instead of timing out.

```
WsrmUtils.setFinalMessage((Stub)port);
```

- Using the stub you annotated with the `@ServiceClient` annotation, invoke the `echo` reliable operation:

```
port.echo(input2);
```

- Create methods to handle the asynchronous response or failure. Use the `@weblogic.jws.AsyncResponse` and `@weblogic.jws.AsyncFailure` annotations:

```
@AsyncResponse(target = "port", operation = "echo")
public void onEchoAsyncResponse(String msg) {
  System.out.println("ClientService: Got async response for request : " + msg);
  responseMessage = msg;
}

@AsyncFailure(target = "port", operation = "echo")
public void onEchoAsyncFailure(Throwable t) {
  System.out.println("ClientService: Got async FAILURE for request : " + t);
  t.printStackTrace();
}
```

For more information about generating asynchronous response and failure methods, see Section 7.4, "Writing the Asynchronous JWS File".

When programming the client Web service:

- Do not specify any reliable messaging annotations (other than `@ReliabilityErrorHandler`) or use any reliable messaging assertions in the associated WS-Policy files.
- Do not specify the `wsdlLocation` attribute of the `@ServiceClient` annotation. This is because the runtime retrieval of the specified WSDL might not succeed; therefore, it is better for WebLogic Server to use a local WSDL file instead.

8.9 Updating the build.xml File for a Client of a Reliable Web Service

To update a `build.xml` file to generate the JWS file that invokes the operation of a reliable Web service, add `taskdef` and a `build-reliable-client` targets similar to the following:

```
<taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JwscTask" />

<target name="build-reliable-client">

    <jwsc
        enableAsyncService="true"
        srcdir="src"
        destdir="${client-ear-dir}" >

        <jws file="examples/webservices/reliable/ReliableClientImpl.java">

          <clientgen

wsdl="http://${wls.destination.host}:${wls.destination.port}/ReliableHelloWorld/Re
liableHelloWorld?WSDL"
            packageName="examples.webservices.reliable"/>

        </jws>

    </jwsc>

</target>
```

Use the `taskdef` Ant task to define the full classname of the `jwsc` Ant tasks.

Update the `jwsc` Ant task that compiles the client Web service to include a `<clientgen>` child element of the `<jws>` element so as to generate and compile the JAX-RPC stubs for the deployed `ReliableHelloWorld` Web service. The `jwsc` Ant

task automatically packages them in the generated WAR file so that the client Web service can immediately access the stubs. You do this because the `ReliableClientImpl` JWS file imports and uses one of the generated classes.

8.10 Using Reliable Messaging With MTOM

The following example shows a simple JWS file that implements a reliable Web service and uses MTOM/XOP to optimize the transmission of XML data of type `xs:base64Binary` in SOAP messages; see the explanation after the example for coding guidelines that correspond to the Java code in **bold**. This example builds on the example provided in Section 8.8, "Programming Guidelines for the JWS File That Invokes a Reliable Web Service".

```
package examples.webservices.reliable;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.Oneway;
import weblogic.jws.WLHttpTransport;

import weblogic.jws.ReliabilityBuffer;
import weblogic.jws.BufferQueue;
import weblogic.jws.Policy;
import weblogic.jws.Policies;

/**
 * Simple reliable Web Service.
 */

@WebService(name="ReliableHelloWorldPortType",
             serviceName="ReliableHelloWorldService")

@WLHttpTransport(contextPath="ReliableHelloWorld",
      serviceUri="ReliableHelloWorld",
      portName="ReliableHelloWorldServicePort")

@Policies({@Policy(uri="ReliableHelloWorldPolicy.xml",
           direction=Policy.Direction.both,
           attachToWsdl=true),
           @Policy(uri = "policy:Mtom.xml")})

@BufferQueue(name="webservices.reliable.queue")

public class ReliableHelloWorldImpl {

  @WebMethod()
  @Oneway()
  @ReliabilityBuffer(retryCount=10, retryDelay="10 seconds")

  public void helloWorld(String input) {
    System.out.println(" Hello World " + input);

  }

  @WebMethod

  public byte[] echoBinary(byte[] bytes) {
    return bytes;
  }
```

```
}
```

As illustrated in the previous example (in **bold** text), follow these guidelines when programming the JWS file that invokes a reliable Web service with MTOM:

- Use the "@weblogic.jws.Policy" annotation to specify that the pre-packaged `Mtom.xml` file should be applied to your Web service. Use the "@weblogic.jws.Policies" annotation to group multiple WS-Policy files, including the reliable messaging policy file and the MTOM policy file.

```
@Policies({@Policy(uri="ReliableHelloWorldPolicy.xml",
           direction=Policy.Direction.both,
           attachToWsdl=true),
           @Policy(uri = "policy:Mtom.xml")})
```

- Use the Java `byte[]` data type in your Web service operations as either a return value or input parameter whenever you want the resulting SOAP message to use MTOM/XOP to send or receive the binary data.

```
public byte[] echoBinary(byte[] bytes) {
   return bytes;
}
```

> **Note:** In this release of WebLogic Server, the only supported Java data type when using MTOM/XOP is `byte[]`; other binary data types, such as `image`, are not supported.

8.11 Client Considerations When Redeploying a Reliable Web Service

WebLogic Server supports production redeployment, which means that you can deploy a new version of an updated reliable WebLogic Web service alongside an older version of the same Web service.

WebLogic Server automatically manages client connections so that only *new* client requests are directed to the new version. Clients already connected to the Web service during the redeployment continue to use the older version of the service until they complete their work, at which point WebLogic Server automatically retires the older Web service. If the client is connected to a reliable Web service, its work is considered complete when the existing reliable message sequence is explicitly ended by the client or as a result of a timeout.

For additional information about production redeployment and Web service clients, see "Client Considerations When Redeploying a Web Service" on page 6-17.

8.12 Using Reliable Messaging With a Proxy Server

Client applications that invoke reliable Web services might not invoke the operation directly, but rather, use a proxy server. Reasons for using a proxy include the presence of a firewall or the deployment of the invoked Web service to a cluster.

In this case, the WebLogic Server instance that hosts the invoked Web service must be configured with the address and port of the proxy server. If your Web service is deployed to a cluster, you must configure every server in the cluster.

This procedure describes how to create a network channel, the primary configurable WebLogic Server resource for managing network connection. Network channels enable you to provide a consistent way to access the front-end address of a cluster. For

more information about network channels, see "Understanding Network Channels" in *Administering Server Environments for Oracle WebLogic Server*.

For each server instance:

1. Create a network channel for the protocol you use to invoke the Web service. You must name the network channel `weblogic-wsee-proxy-channel-`*`XXX`*, where *`XXX`* refers to the protocol. For example, to create a network channel for HTTPS, call it `weblogic-wsee-proxy-channel-https`.

   See "Configure custom network channels" in *Oracle WebLogic Server Administration Console Online Help* for general information about creating a network channel.

2. Configure the network channel, updating the **External Listen Address** and **External Listen Port** fields with the address and port of the proxy server, respectively.

3. Disable the asynchronous response service on the WebLogic Server proxy server by starting the WebLogic Server instance using the `-Dweblogic.wsee.skip.async.response=true` Java system property.

   By default, every WebLogic Server instance deploys an internal asynchronous Web service that handles the asynchronous request-response feature. If you do not specify this system property, asynchronous messages will never get to the cluster, as required, because the asynchronous service on the proxy server will consume them instead.

9

Creating Conversational Web Services

This chapter describes how to create conversational WebLogic Java API for XML-based RPC (JAX-RPC) Web services.

This chapter includes the following topics:

9.1 Overview of Conversational Web Services

A Web service and the client application that invokes it may communicate multiple times to complete a single task. Also, multiple client applications might communicate with the same Web service at the same time. *Conversations* provide a straightforward way to keep track of data between calls and to ensure that the Web service always responds to the correct client.

Conversations meet two challenges inherent in persisting data across multiple communications:

- Conversations uniquely identify a two-way communication between one client application and one Web service so that messages are always returned to the correct client. For example, in a shopping cart application, a conversational Web service keeps track of which shopping cart belongs to which customer. A conversational Web service implements this by creating a unique conversation ID each time a new conversation is started with a client application.
- Conversations maintain state between calls to the Web service; that is, they keep track of the data associated with a particular client application between its calls to the service. Conversations ensure that the data associated with a particular client

is saved until it is no longer needed or the operation is complete. For example, in a shopping cart application, a conversational Web service remembers which items are in the shopping cart while the customer continues shopping. Maintaining state is also needed to handle failure of the computer hosting the Web service in the middle of a conversation; all state-related data is persisted to disk so that when the computer comes up it can continue the conversation with the client application.

WebLogic Server manages this unique ID and state by creating a conversation context each time a client application initiates a new conversation. The Web service then uses the context to correlate calls to and from the service and to persist its state-related data.

Conversations between a client application and a Web service have three distinct phases:

- Start—A client application initiates a conversation by invoking the start operation of the conversational Web service. The Web service in turn creates a new conversation context and an accompanying unique ID, and starts an internal timer to measure the idle time and the age of the conversation.
- Continue—After the client application has started the conversation, it invokes one or more continue operations to continue the conversation. The conversational Web service uses the ID associated with the invoke to determine which client application it is conversing with, what state to persist, and which idle timer to reset. A typical continue operation would be one that requests more information from the client application, requests status, and so on.
- Finish—A client application explicitly invokes the finish operation when it has finished its conversation; the Web service then marks any data or resources associated with the conversation as deleted.

Conversations typically occur between two WebLogic Web services: one is marked conversational and defines the start, continue, and finish operations and the other Web service uses the `@ServiceClient` annotation to specify that it is a client of the conversational Web service. You can also invoke a conversational Web service from a stand-alone Java client, although there are restrictions.

As with other WebLogic Web service features, you use JWS annotations to specify that a Web service is conversational.

Note: The client Web service that invokes a conversational Web service is not required to also be conversational. However, if the client is *not* conversational, there is a danger of multiple instances of this client accessing the same conversational Web service stub and possibly corrupting the saved conversational state. If you believe this might true in your case, then specify that the client Web service also be conversational. In this case you cannot use a stand-alone Java client, because there is no way to mark it as conversational using the WebLogic APIs.

A conversational Web service on its own does not guarantee message delivery or that the messages are delivered in order, exactly once. If you require this kind of message delivery guarantee, you must also specify that the Web service be reliable. See Section 8.2, "Using Web Service Reliable Messaging: Main Steps" and Chapter 11, "Using the Asynchronous Features Together."

9.2 Creating a Conversational Web Service: Main Steps

The following procedure describes how to create a conversational Web service, as well as a client Web service and stand-alone Java client application, both of which initiate and conduct a conversation. The procedure shows how to create the JWS files that implement the two Web services from scratch. If you want to update existing JWS files, you can also use this procedure as a guide.

It is assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file to which you can add targets for running the `jwsc` Ant task and deploying the generated conversational Web service. It is further assumed that you have a similar setup for the WebLogic Server instance that hosts the client Web service that initiates the conversation. For more information, see the following sections:

- "Examples for JAX-RPC Web Service Developers" on page 2-1
- "Developing JAX-RPC Web Services" on page 3-1
- "Programming the JWS File" on page 4-1
- "Developing JAX-RPC Web Service Clients" on page 6-1

Table 9–1 Steps to Create a Conversational Web Service

#	Step	Description
1	Create a new JWS file, or update an existing one, that implements the conversational Web service.	Use your favorite IDE or text editor. See Section 9.3, "Programming Guidelines for the Conversational JWS File".
2	Update your `build.xml` file to include a call to the `jwsc` Ant task to compile the conversational JWS file into a Web service.	See "Running the jwsc WebLogic Web Services Ant Task" on page 3-7.
3	Run the Ant target to build the conversational Web service.	For example: `prompt> ant build-mainService`
4	Deploy the target Web service as usual.	See "Deploying and Undeploying WebLogic Web Services" on page 3-14.
5	Create a new JWS file, or update an existing one, that implements the client Web service.	If the client application is a stand-alone Java client, see Section 9.7, "Updating a Stand-Alone Java Client to Invoke a Conversational Web Service". Skip Steps 6-9. If the client application is itself a Web service, follow Steps 6-9.
6	Create a new JWS file, or update an existing one, that initiates and conducts the conversation with the conversational Web service.	Use your favorite IDE or text editor. It is assumed that the client Web service is deployed to a different WebLogic Server instance form the one that hosts the conversational Web service. See Section 9.4, "Programming Guidelines for the JWS File That Invokes a Conversational Web Service".
7	Update the `build.xml` file that builds the client Web service.	See Section 9.6, "Updating the build.xml File for a Client of a Conversational Web Service".
8	Run the Ant target to build the client Web services.	For example: `prompt> ant build-clientService`
9	Deploy the client Web service as usual.	See "Deploying and Undeploying WebLogic Web Services" on page 3-14.

9.3 Programming Guidelines for the Conversational JWS File

The following example shows a simple JWS file that implements a conversational Web service; see the explanation after the example for coding guidelines that correspond to the Java code in bold.

```
package examples.webservices.conversation;

import java.io.Serializable;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.Conversation;
import weblogic.jws.Conversational;
import weblogic.jws.Context;
import weblogic.wsee.jws.JwsContext;
import weblogic.wsee.jws.ServiceHandle;

import javax.jws.WebService;
import javax.jws.WebMethod;
@Conversational(maxIdleTime="10 minutes",
                maxAge="1 day",
                runAsStartUser=false,
                singlePrincipal=false )
@WebService(name="ConversationalPortType",
            serviceName="ConversationalService",
            targetNamespace="http://examples.org/")

@WLHttpTransport(contextPath="conv",
                 serviceUri="ConversationalService",
                 portName="ConversationalServicePort")

/**
 * Conversational Web service.
 */

public class ConversationalServiceImpl implements Serializable {

  @Context
  private JwsContext ctx;
  public String status = "undefined";

  @WebMethod
  @Conversation (Conversation.Phase.START)
  public String start() {

    ServiceHandle handle = ctx.getService();
    String convID = handle.getConversationID();

    status = "start";
    return "Starting conversation, with ID " + convID + " and status equal to " + status;

  }

  @WebMethod
  @Conversation (Conversation.Phase.CONTINUE)
  public String middle(String message) {

    status = "middle";
    return "Middle of conversation; the message is: " + message + " and status is " + status;

  }
```

```
  @WebMethod
  @Conversation (Conversation.Phase.FINISH)
  public String finish(String message ) {

    status = "finish";
    return "End of conversation; the message is: " + message + " and status is " + status;

  }

}
```

Follow these guidelines when programming the JWS file that implements a conversational Web service. Code snippets of the guidelines are shown in bold in the preceding example.

- Conversational Web services must implement `java.io.Serializable`, so you must first import the class into your JWS file:

  ```
  import java.io.Serializable;
  ```

- Import the conversational JWS annotations:

  ```
  import weblogic.jws.Conversation;
  import weblogic.jws.Conversational;
  ```

- If you want to access runtime information about the conversational Web service, import the `@Context` annotation and context APIs:

  ```
  import weblogic.jws.Context;

  import weblogic.wsee.jws.JwsContext;
  import weblogic.wsee.jws.ServiceHandle;
  ```

- Use the class-level `@Conversational` annotation to specify that the Web service is conversational. Although this annotation is optional (assuming you *are* specifying the `@Conversation` method-level annotation), it is a best practice to always use it in your JWS file to clearly specify that your Web service is conversational.

  Specify any of the following optional attributes: `maxIdleTime` is the maximum amount of time that the Web service can be idle before WebLogic Server finishes the conversation; `maxAge` is the maximum age of the conversation; `runAsStartUser` indicates whether the continue and finish phases of an existing conversation are run as the user who started the conversation; and `singlePrincipal` indicates whether users other than the one who started a conversation are allowed to execute the continue and finish phases of the conversation.

  ```
  @Conversational(maxIdleTime="10 minutes",
                  maxAge="1 day",
                  runAsStartUser=false,
                  singlePrincipal=false )
  ```

  If a JWS file includes the `@Conversational` annotation, all operations of the Web service are conversational. The default phase of an operation, if it does not have an explicit `@Conversation` annotation, is continue. However, because a conversational Web service is required to include at least one start and one finish operation, you *must* use the method-level `@Conversation` annotation to specify which methods implement these operations.

See "weblogic.jws.Conversational" in *WebLogic Web Services Reference for Oracle WebLogic Server* for additional information and default values for the attributes.

- Your JWS file must implement `java.io.Serializable`:

```
public class ConversationalServiceImpl implements Serializable {
```

- To access runtime information about the Web service, annotate a private class variable, of data type `weblogic.wsee.jws.JwsContext`, with the field-level `@Context` JWS annotation:

```
@Context
private JwsContext ctx;
```

- Use the `@Conversation` annotation to specify the methods that implement the start, continue, and finish phases of your conversation. A conversation is required to have at least one start and one finish operation; the continue operation is optional. Use the following parameters to the annotation to specify the phase: `Conversation.Phase.START`, `Conversation.Phase.CONTINUE`, or `Conversation.Phase.FINISH`. The following example shows how to specify the start operation:

```
@WebMethod
@Conversation (Conversation.Phase.START)
public String start() {...
```

  If you mark just one method of the JWS file with the `@Conversation` annotation, then the entire Web service becomes conversational and each operation is considered part of the conversation; this is true even if you have not used the optional class-level `@Conversational` annotation in your JWS file. Any methods not explicitly annotated with `@Conversation` are, by default, continue operations. This means that, for example, if a client application invokes one of these continue methods without having previously invoked a start operation, the Web service returns a runtime error.

  Finally, if you plan to invoke the conversational Web service from a stand-alone Java client, the start operation is required to be request-response, or in other words, it *cannot* be annotated with the `@Oneway` JWS annotation. The operation can return `void`. If you are going to invoke the Web service only from client applications that run in WebLogic Server, then this requirement does not apply.

  See "weblogic.jws.Conversation" in *WebLogic Web Services Reference for Oracle WebLogic Server* for additional information.

- Use the `JwsContext` instance to get runtime information about the Web service.

  For example, the following code in the start operation gets the ID that WebLogic Server assigns to the new conversation:

```
ServiceHandle handle = ctx.getService();
String convID = handle.getConversationID();
```

9.4 Programming Guidelines for the JWS File That Invokes a Conversational Web Service

The following example shows a simple JWS file for a Web service that invokes the conversational Web service described in Section 9.3, "Programming Guidelines for the Conversational JWS File"; see the explanation after the example for coding guidelines that correspond to the Java code in bold.

```
package examples.webservices.conversation;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;

import weblogic.wsee.conversation.ConversationUtils;

import javax.jws.WebService;
import javax.jws.WebMethod;

import javax.xml.rpc.Stub;

import examples.webservices.conversation.ConversationalPortType;

import java.rmi.RemoteException;

@WebService(name="ConversationalClientPortType",
            serviceName="ConversationalClientService",
            targetNamespace="http://examples.org/")

@WLHttpTransport(contextPath="convClient",
                 serviceUri="ConversationalClient",
                 portName="ConversationalClientPort")

/**
 *  client that has a conversation with the ConversationalService.
 */

public class ConversationalClientImpl {

  @ServiceClient(
     wsdlLocation="http://localhost:7001/conv/ConversationalService?WSDL",
     serviceName="ConversationalService",
     portName="ConversationalServicePort")

  private ConversationalPortType port;

  @WebMethod
  public void runConversation(String message) {

    try {

      // Invoke start operation
      String result = port.start();
      System.out.println("start method executed.");
      System.out.println("The message is: " + result);

      // Invoke continue operation
      result = port.middle(message );
      System.out.println("middle method executed.");
      System.out.println("The message is: " + result);

      // Invoke finish operation
      result = port.finish(message );
      System.out.println("finish method executed.");
      System.out.println("The message is: " + result);
      ConversationUtils.renewStub((Stub)port);

    }
    catch (RemoteException e) {
```

```
      e.printStackTrace();
    }

  }

}
```

Follow these guidelines when programming the JWS file that invokes a conversational Web service; code snippets of the guidelines are shown in bold in the preceding example:

- Import the `@ServiceClient` JWS annotation:

  ```
  import weblogic.jws.ServiceClient;
  ```

- Optionally import the WebLogic utility class for further configuring a conversation:

  ```
  import weblogic.wsee.conversation.ConversationUtils;
  ```

- Import the JAX-RPC stub of the port type of the conversational Web service you want to invoke. The actual stub itself will be created later by the `jwsc` Ant task. The stub package is specified by the `packageName` attribute of the `<clientgen>` child element of `<jws>`, and the name of the stub is determined by the WSDL of the invoked Web service.

  ```
  import examples.webservices.conversation.ConversationalPortType;
  ```

- In the body of the JWS file, use the `@ServiceClient` JWS annotation to specify the WSDL, name, and port of the conversational Web service you want to invoke. You specify this annotation at the field-level on a private variable, whose data type is the JAX-RPC port type of the Web service you are invoking.

  ```
  @ServiceClient(
       wsdlLocation="http://localhost:7001/conv/ConversationalService?WSDL",
       serviceName="ConversationalService",
       portName="ConversationalServicePort")

    private ConversationalPortType port;
  ```

- Using the stub you annotated with the `@ServiceClient` annotation, invoke the start operation of the conversational Web service to start the conversation. You can invoke the start method from any location in the JWS file (constructor, method, and so on):

  ```
  String result = port.start();
  ```

- Optionally invoke the continue methods to continue the conversation. Be sure you use the same stub instance so that you continue the same conversation you started:

  ```
  result = port.middle(message );
  ```

- Once the conversation is completed, invoke the finish operation so that the conversational Web service can free up the resources it used for the current conversation:

  ```
  result = port.finish(message );
  ```

- If you want to reuse the Web service conversation stub to start a new conversation, you must explicitly renew the stub using the `renewStub()` method of the `weblogic.wsee.conversation.ConversationUtils` utility class:

```
ConversationUtils.renewStub((Stub)port);
```

Note: The client Web service that invokes a conversational Web service is not required to also be conversational. However, if the client is *not* conversational, there is a danger of multiple instances of this client accessing the same conversational Web service stub and possibly corrupting the saved conversational state. If you believe this might true in your case, then specify that the client Web service also be conversational.

9.5 ConversationUtils Utility Class

WebLogic Server provides a utility class for use with the conversation feature. Use this class to perform common tasks such as getting and setting the conversation ID and setting configuration options. Some of these tasks are performed in the conversational Web service, some are performed in the client that invokes the conversational Web service. See Section 9.4, "Programming Guidelines for the JWS File That Invokes a Conversational Web Service" for an example of using this class.

See "weblogic.wsee.conversation.ConversationUtils" in *Java API Reference for Oracle WebLogic Server* for details.

9.6 Updating the build.xml File for a Client of a Conversational Web Service

You update a `build.xml` file to generate the JWS file that invokes a conversational Web service by adding `taskdefs` and a `build-clientService` target that looks something like the following example. See the description after the example for details.

```
<taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JwscTask" />

  <target name="build-clientService">

    <jwsc
        enableAsyncService="true"
        srcdir="src"
        destdir="${clientService-ear-dir}" >

        <jws
file="examples/webservices/conversation/ConversationalClientImpl.java" >
          <clientgen

wsdl="http://${wls.hostname}:${wls.port}/conv/ConversationalService?WSDL"
            packageName="examples.webservices.conversation"/>

        </jws>

    </jwsc>

  </target>
```

Use the `taskdef` Ant task to define the full classname of the `jwsc` Ant tasks.

Update the `jwsc` Ant task that compiles the client Web service to include a `<clientgen>` child element of the `<jws>` element so as to generate and compile the JAX-RPC stubs for the deployed `ConversationalService` Web service. The `jwsc`

Ant task automatically packages them in the generated WAR file so that the client Web service can immediately access the stubs. You do this because the `ConversationalClientImpl` JWS file imports and uses one of the generated classes.

9.7 Updating a Stand-Alone Java Client to Invoke a Conversational Web Service

The following example shows a simple stand-alone Java client that invokes the conversational Web service described in Section 9.3, "Programming Guidelines for the Conversational JWS File". See the explanation after the example for coding guidelines that correspond to the Java code in bold.

```
package examples.webservices.conv_standalone.client;

import java.rmi.RemoteException;

import javax.xml.rpc.ServiceException;
import javax.xml.rpc.Stub;
import weblogic.wsee.jaxrpc.WLStub;

/**
 * stand-alone client that invokes and converses with ConversationlService.
 */

public class Main {

  public static void main(String[] args)
      throws ServiceException, RemoteException{

      ConversationalService service = new ConversationalService_Impl(args[0] + "?WSDL");
      ConversationalPortType port = service.getConversationalServicePort();

      // Set property on stub to specify that client is invoking a Web service
      // that uses advanced features; this property is automatically set if
      // the client runs in a WebLogic Server instance.

      Stub stub = (Stub)port;
      stub._setProperty(WLStub.COMPLEX,  "true");

      // Invoke start operation to begin the conversation
      String result = port.start();
      System.out.println("start method executed.");
      System.out.println("The message is: " + result);

      // Invoke continue operation
      result = port.middle("middle" );
      System.out.println("middle method executed.");
      System.out.println("The message is: " + result);

      // Invoke finish operation
      result = port.finish("finish" );
      System.out.println("finish method executed.");
      System.out.println("The message is: " + result);

  }

}
```

Follow these guidelines when programming the stand-alone Java client that invokes a conversational Web service. Code snippets of the guidelines are shown in bold in the preceding example.

- Import the `weblogic.wsee.jaxrpc.WLStub` class:

  ```
  import weblogic.wsee.jaxrpc.WLStub;
  ```

- Set the `WLStub.Complex` property on the JAX-RPC stub of the `ConversationalService` using the `_setProperty` method:

  ```
  Stub stub = (Stub)port;
  stub._setProperty(WLStub.COMPLEX,  "true");
  ```

  This property specifies to the Web services runtime that the client is going to invoke an advanced Web service, in this case a conversational one. This property is automatically set when invoking a conversational Web service from another WebLogic Web service.

- Invoke the start operation of the conversational Web service to start the conversation:

  ```
  String result = port.start();
  ```

- Optionally invoke the continue methods to continue the conversation:

  ```
  result = port.middle(message );
  ```

- Once the conversation is completed, invoke the finish operation so that the conversational Web service can free up the resources it used for the current conversation:

  ```
  result = port.finish(message );
  ```

9.8 Example Conversational Web Service .NET Client

This section demonstrates how to create a .NET WSE3.0 client for a WebLogic conversational Web service. The example includes the following files:

- `ConversationService.java` -- JWS file that uses the `@Conversation` and `@Callback` annotations to implement a conversational Web service. `ConversationService.java` can optionally communicate results to its client via a callback.
- `Service.cs` -- The C# source file of the `ConversationClient` .NET Web service that acts as a client to the `ConversationService` Web service.

  The sample .NET client can participate in conversations with `ConversationService`, as well as receiving results via callback.
- build.xml -- Ant build file that contains targets for building and deploying the Conversational Web service.

These files are described in detail in the sections that follow.

9.8.1 ConversationService.java File

The example `ConversationService.java` file is shown in Example 9–1. The example includes extensive comments that describe its function.

Example 9–1 ConversationService.java File

```
package conv;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.Oneway;

import weblogic.jws.Conversation;
import weblogic.jws.Callback;
import weblogic.jws.CallbackService;

import java.io.Serializable;

/**
 * Demonstrates use of the @Conversation annotation to manage the lifetime of the
service
 * and provide data persistence and message correlation.
 *
 * Remember that multiple clients might invoke a web service simultaneously.  When
the
 * web service stores data relevant to the client or calls additional services
 * in order to process a client's request, the service must be able to process
returned
 * data from the external services in the context of the specific client it
relates
 * to.  This is all automatic when conversations are used.
 *
 * Remember that not all clients are capable of accepting callbacks.
Specifically,
 * clients operating from behind firewalls may not be able to receive asynchronous
 * callbacks.  You may wish to provide a synchronous interface, like this one,
 * for such clients.  If a client can accept callbacks, it must send a callback
endpoint refrence
 * as part of any "start conversation" method invocation.
 *
 * To see the behavior in the Test View, invoke startRequest and then
getRequestStatus
 * several times quickly.
 */
@WebService(serviceName = "ConversationService", portName = "conversation",
targetNamespace = "http://www.openuri.org/")
public class ConversationService implements Serializable {

  @Callback
  public CallbackInterface callback;
  private boolean useCallbacks;
  private int num;

  /**
   * Starts the conversation
   */
  @Conversation(Conversation.Phase.START)
  @WebMethod
  public void startRequest(boolean useCallbacks) {
    this.useCallbacks = useCallbacks;
  }

  @WebMethod
  @Conversation(Conversation.Phase.CONTINUE)
  public String getRequestStatus() {
```

```
    num++;
    if (num == 1)
      return "This is the first time you call getRequestStatus method.";
    if (num == 2 && useCallbacks) {
      callback.onResultReady("finished");
      return "This is the second time you call  getRequestStatus method, the
conversation has been terminated automtically when the onResultReady callback
method is invoked.";
    } else
      return "You have called getRequestStatus method " + num + " times";

  }

  /**
   * Used to tell Conversation.jws that the current conversation is
   * no longer needed.
   */
  @WebMethod
  @Conversation(Conversation.Phase.FINISH)
  public void terminateRequest() {

  }

  @CallbackService(serviceName = "ConversationCallbackService")
  public interface CallbackInterface {

    /**
     * Callback to invoke on the client when the external service
     * returns its result. Will only be called it the client can
     * accept callbacks and told us where to send them.
     * <p/>
     * If this callback is used, it implicitly terminates the
     * conversation with no action required on the part of the
     * client.
     */
    @WebMethod
    @Oneway
    @Conversation(Conversation.Phase.FINISH)
    public void onResultReady(String result);
  }

}
```

9.8.2 Service.cs File

The example `Service.cs` file is shown in Example 9–2.

This conversation proxy file was created using the Microsoft WSDL to Proxy Class tool `WseWsdl3.exe` (see http://msdn.microsoft.com/en-us/library/aa529578.aspx) and the `ConversationService` Web service's WSDL file.

The example includes extensive comments that describe its function.

Example 9–2 Service.cs File

```
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
```

```
using System.Diagnostics;
using System.IO;
using System.Xml;
using Microsoft.Web.Services3.Addressing;
using Microsoft.Web.Services3;
using System.Collections.Generic;
using Microsoft.Web.Services3.Design;


[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class Service : System.Web.Services.WebService
{

    public Service () {

        //Uncomment the following line if using designed components
        //InitializeComponent();
    }

    /*
    * start invokes the Conversation web service's startRequest
    * operation.
    *
    * Since the Conversation web service is conversational,
    * we must also persist the ReplyTo endpoint reference SOAP header
    * for subsequent calls.
    *
    * Since the Conversation web service can optionally communicate
    * the result of it's work via a callback, we must prepare a
    * second SOAP header CallbackTo SOAP header, which is the endpoint reference
    * of the recipient to which callbacks should be sent.
    */
    [WebMethod(EnableSession = true)]
    public void start(Boolean useCallbacks, Boolean useIPAddress)
    {
        /*
         * The Conversation proxy was created using .NET WSE3.0's WseWsdl3.exe
         * application and the Conversation.jws's WSDL file.  The WSDL
         * file for any WLS web service may be obtained
         * by hitting the web service's URL with "?WSDL" appended to
         * the end.  For example:
         *
         * http://somehost:7001/samples/async/Conversation.jws?WSDL
         *
         * WseWsdl3.exe produces a C# proxy class.  Place the resulting
         * ConversationService.cs file in your .NET project, then use Visual
         * Studio's Project->Add Existing Item menu action to "import"
         * the class into the project.
         */
        ConversationServiceSoapBinding conv;
        String callbackLocation;
        int asmxIndex;

        /*
         * Construct the callback URL from various pieces of
         * server and HttpRequest info.
         */
        Uri requestUrl = Context.Request.Url;
```

```
        if (useIPAddress)
        {
            /*
             * if useIPAddress is true, construct the callback address
             * with the IP address of this host.
             */
            callbackLocation = requestUrl.Scheme + "://" +
System.Net.Dns.GetHostByName(requestUrl.Host).AddressList[0] +
                ":" + requestUrl.Port + requestUrl.AbsolutePath;
        }
        else
        {
            /*
             * if useIPAddress is false, construct the callback address
             * with the hostname of this host.
             */
            callbackLocation = requestUrl.Scheme + "://" + requestUrl.Host +
                ":" + requestUrl.Port + requestUrl.AbsolutePath;
        }

        // Remove everything after ".asmx"
        asmxIndex = callbackLocation.IndexOf(".asmx") + 5;
        callbackLocation = callbackLocation.Remove(asmxIndex,
                                callbackLocation.Length - asmxIndex);

        /*
         * Create an instance of the proxy for the Conversation
         * web service.
         *
         */
        conv = new ConversationServiceSoapBinding();


        /*
         * When callback is enabled, a custom callback header should be added into
         * the outbound soap message.
         *
         */
        if (useCallbacks)
            enableSoapFilterToAddCallbackHeader(conv, callbackLocation);


        /*
         * Invoke the startRequest method of the web service. The
         * single boolean parameter determines whether the Conversation
         * web service will use callbacks to communicate the result
         * back to this client.
         *
         * If the argument is true, an onResultReady callback will
         * be sent when the result is ready.  This client must implement
         * a method with that name that expects the message shape defined
         * by the target web service (returns void and accepts a single
         * string argument).  See the onResultReady method below.
         *
         * If the argument to startRequest is false, callbacks will not
         * be used and this client must use the getRequestStatus method
         * to poll the Conversation web service for the result.
         */
        conv.startRequest(useCallbacks);
        /*
```

```
         * Persist the ReplyTo header  in session state so that it can
         * be used in other methods that take part in the conversation.
         *
         * This is not safe since one session could start multiple
         * conversations, but there is no other apparent way to persist
         * this information.  Member variables of WebService classes
         * are not persisted across method invocations.
         */
        Session["ConversationReplyTo"] =
conv.ResponseSoapContext.Addressing.ReplyTo;


    }

    /* the CallbackTo header defintion isn't exposed by WLS9x/WLS10x callback
service.
     * So we need to use SOAPFilter to add the CallbackTo header.
     *
     */
    private static void
enableSoapFilterToAddCallbackHeader(ConversationServiceSoapBinding conv, String
callbackLocation)
    {
        //Create a custom policy.
        Policy myPolicy = new Policy();
        // Create a new policy assertion
        MyPolicyAssertion myAssertion = new MyPolicyAssertion(callbackLocation);
        // Add the assertion to the policy
        myPolicy.Assertions.Add(myAssertion);
        //Set the custom policy you have created on the client proxy
        conv.SetPolicy(myPolicy);
    }

     /*
     * getStatus invokes Conversation's getRequestStatus method.
     * getRequestStatus is a polling method that is an alternative
     * for web services that cannot recieve callbacks.
     *
     * Note that a conversation must be started with startRequest before
     * this method may be invoked.  If not, or if this method is invoked
     * outside of a conversation for any reason, it will get back a SOAP
     * fault indicating that the conversation does not exist.
     */
    [WebMethod(EnableSession = true)]
    public String getStatus()
    {
        String result;

        /*
         * Create an instance of the proxy for the Conversation
         * web service.  We could probably persist the proxy instance
         * in session state, but chose not to.
         */
        ConversationServiceSoapBinding conv = new
ConversationServiceSoapBinding();

        /*
         * change the destination to the ReplyTo endpoint reference we cached on
session state in
         * the start method.
```

```
         */
        conv.RequestSoapContext.Addressing.Destination =
(EndpointReference)Session["ConversationReplyTo"];
        /*
         * Invoke the getRequestStatus method of the web service.
         */
        result = conv.getRequestStatus();
        return result;
    }

    /*
     * finish invokes Conversation's terminateRequest method, which
     * terminates the current conversation.
     *
     * Note that a conversation must be started with startRequest before
     * this method may be invoked.  If not, or if this method is invoked
     * outside of a conversation for any reason, it will get back a SOAP
     * fault indicating that the conversation does not exist.
     */
    [WebMethod(EnableSession = true)]
    public void finish()
    {
        /*
         * Create an instance of the proxy for the Conversation
         * web service.  We could probably persist the proxy instance
         * in session state, but chose not to.
         */
        ConversationServiceSoapBinding conv = new
ConversationServiceSoapBinding();

        /*
         * change the destination to the ReplyTo endpoint reference we cached on
session state in
         * the start method. Both "continue" and "finish" methods use the same
destination.
         */
        conv.RequestSoapContext.Addressing.Destination =
(EndpointReference)Session["ConversationReplyTo"];
        /*
         * Invoke the terminateRequest method of the web service.
         */
        conv.terminateRequest();
    }

    /*
     * onResultReady is a callback handler for the onResultReady
     * callback that Conversation.jws can optionally use to return
     * its results.
     *
     * .NET WSE3.0 does not support callbacks directly, but a callback is just
     * a method invocation message.  So if you construct a WebMethod with
     * the same signature as the callback and set the XML namespace
     * properly, it serves as a callback handler.
     *
     */
    [WebMethod]
    [SoapDocumentMethod(OneWay = true,
        Action = "http://www.openuri.org/ConversationService_
CallbackInterface/onResultReady",
        RequestElementName = "http://www.openuri.org/",
```

```
        ResponseNamespace = "http://www.openuri.org/"
        )]
    public void onResultReady(String result)
    {
        /*
         * When the callback is invoked, log a message to the
         * hardcoded file c:\temp\ConversationClient.log.
         *
         * Note: if c:\temp does not exist on this server, an
         * Exception will be raised.  Since it is not handled here,
         * it will be returned as a SOAP fault to the Conversation
         * web service.
         */
        TextWriter output;
        output = File.AppendText("c:\\temp\\ConversationClient.log");
        String msg = "[" + DateTime.Now.ToString() + "] callback received";
        output.WriteLine(msg);
        output.Flush();
        output.Close();
    }


}

public class MyFilter : Microsoft.Web.Services3.SoapFilter
{
    private String callbackLocation;

    public MyFilter(String callbackLocation)
    {
        this.callbackLocation = callbackLocation;
    }

    public override SoapFilterResult ProcessMessage(SoapEnvelope envelope)
    {

       //create the CallbackTo soap element.
       XmlDocument xmldoc = new XmlDocument();
       XmlElement xmlEle = xmldoc.CreateElement("callback", "CallbackTo",
"http://www.openuri.org/2006/03/callback");

       //create the CallbackTo endpoint reference.
       Address callbacto = new Address(new Uri(callbackLocation));
       XmlElement xmlEle2 =new EndpointReference(callbacto).GetXml(xmldoc);
       //add the CallbackTo endpoint reference into CallbackTo SOAP element.
       xmlEle.AppendChild(xmlEle2.FirstChild);
       //add the whole CallbackTo SOAP element into SOAP header.
       XmlNode callbackheader = envelope.ImportNode(xmlEle, true);
       envelope.DocumentElement.FirstChild.AppendChild(callbackheader);
       return SoapFilterResult.Continue;

    }
}

public class MyPolicyAssertion : Microsoft.Web.Services3.Design.PolicyAssertion
{
    private String callbackLocation;

    public MyPolicyAssertion(String callbackLocation)
    {
```

```
        this.callbackLocation = callbackLocation;
    }

    public override  SoapFilter CreateClientInputFilter(FilterCreationContext
context)
    {
        return null;
    }


    public override  SoapFilter CreateClientOutputFilter(FilterCreationContext
context)
    {
        //use MyFilter to add the CallbackTo header in the outbound soap message.
        return new MyFilter(callbackLocation);
    }

    public override  SoapFilter CreateServiceInputFilter(FilterCreationContext
context)
    {
        return null;
    }

    public override  SoapFilter CreateServiceOutputFilter(FilterCreationContext
context)
    {
        return null;
    }

}
```

9.8.3 build.xml File

The example `build.xml` file is shown in Example 9–3.

`build.xml` assumes that you copy the example source files to a new directory *EXAMPLES_HOME*`\wl_server\examples\src\examples\webservices\conv`, where *EXAMPLES_HOME* represents the directory in which the WebLogic Server code examples are configured. For more information about the WebLogic Server code examples, see "Sample Applications and Code Examples" in *Understanding Oracle WebLogic Server*.

`build.xml` also requires that you first set your examples environment correctly via *ORACLE_HOME*`\user_projects\domains\wl_server>setExamplesEnv.cmd(sh` and that the examples server is already started.

The example includes comments that describe the build file function and targets.

Example 9–3 build.xml File

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="webservices.conversation" default="all" basedir=".">

  <!-- set global properties for this build -->
  <property file="../../../examples.properties"/>

  <property name="client.dir" value="${client.classes.dir}/webservices_
conversation" />
  <property name="package.dir" value="examples/webservices/conv"/>
  <property name="package" value="examples.webservices.conv"/>
  <property name="ear.dir"
```

```
value="${examples.build.dir}/webservicesConversationEar" />

  <path id="client.class.path">
    <pathelement path="${java.class.path}"/>
  </path>

  <!-- Web service WLS Ant task definitions -->
  <taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JwscTask" />
  <taskdef name="clientgen"
      classname="weblogic.wsee.tools.anttasks.ClientGenTask" />

  <target name="all" depends="build, deploy"/>


  <target name="clean">
    <delete dir="${ear.dir}"/>
  </target>

  <!-- Target that builds the conversational  Web service -->
  <target name="build" description="Target that builds the MTOM Web service">
    <jwsc
      srcdir="${examples.src.dir}/${package.dir}"
      sourcepath="${examples.src.dir}"
      destdir="${ear.dir}"
      classpath="${java.class.path}"
      keepGenerated="true"
      deprecation="${deprecation}"
      debug="${debug}">
        <jws file="ConversationService.java">
<WLHttpTransport contextPath="/samples/async" serviceURI="conversation.jws"/>
</jws>
    </jwsc>
  </target>



  <!-- Target that deploys the conversational Web service -->
  <target name="deploy" description="Target that deploys the conversational Web
service">
    <wldeploy
      action="deploy"
      source="${ear.dir}"
      user="${wls.username}"
      password="${wls.password}"
      verbose="true"
      adminurl="t3://${wls.hostname}:${wls.port}"
      targets="${wls.server.name}"
      failonerror="${failondeploy}"/>
  </target>

  <!-- Target that undeploys the conversational Web service -->
  <target name="undeploy" description="Target that deploys the conversational Web
service">
    <wldeploy
      action="undeploy"
      name="webservicesConversationEar"
      user="${wls.username}"
      password="${wls.password}"
      verbose="true"
```

```
      adminurl="t3://${wls.hostname}:${wls.port}"
      targets="${wls.server.name}"
      failonerror="${failondeploy}"/>
  </target>

</project>
```

9.9 Client Considerations When Redeploying a Conversational Web Service

WebLogic Server supports production redeployment, which means that you can deploy a new version of an updated conversational WebLogic Web service alongside an older version of the same Web service.

WebLogic Server automatically manages client connections so that only *new* client requests are directed to the new version. Clients already connected to the Web service during the redeployment continue to use the older version of the service until they complete their work, at which point WebLogic Server automatically retires the older Web service. If the client is connected to a conversational Web service, its work is considered complete when the existing conversation is explicitly ended by the client or because of a timeout.

For additional information about production redployment and Web service clients, see "Client Considerations When Redeploying a Web Service" on page 6-17.

10 Creating Buffered Web Services

This chapter describes how to create buffered WebLogic Java API for XML-based RPC (JAX-RPC) Web services.

This chapter includes the following topics:

- Section 10.1, "Overview of Buffered Web Services"
- Section 10.2, "Creating a Buffered Web Service: Main Steps"
- Section 10.3, "Configuring the Host WebLogic Server Instance for the Buffered Web Service"
- Section 10.4, "Programming Guidelines for the Buffered JWS File"
- Section 10.5, "Programming the JWS File That Invokes the Buffered Web Service"
- Section 10.6, "Updating the build.xml File for a Client of the Buffered Web Service"

10.1 Overview of Buffered Web Services

When a buffered operation is invoked by a client, the method operation goes on a JMS queue and WebLogic Server deals with it asynchronously. As with Web service reliable messaging, if WebLogic Server goes down while the method invocation is still in the queue, it will be dealt with as soon as WebLogic Server is restarted. When a client invokes the buffered Web service, the client does not wait for a response from the invoke, and the execution of the client can continue.

10.2 Creating a Buffered Web Service: Main Steps

The following procedure describes how to create a buffered Web service and a client Web service that invokes an operation of the buffered Web service. The procedure shows how to create the JWS files that implement the two Web services from scratch. If you want to update existing JWS files, use this procedure as a guide. The procedure also shows how to configure the WebLogic Server instance that hosts the buffered Web service.

> **Note:** Unless you are also using the asynchronous request-response feature, you do not need to invoke a buffered Web service from another Web service, you can also invoke it from a stand-alone Java application.

It is assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file to which you can add targets for running the

`jwsc` Ant task and deploying the generated buffered Web service. It is further assumed that you have a similar setup for the WebLogic Server instance that hosts the client Web service that invokes the buffered Web service. For more information, see the following sections:

- "Examples for JAX-RPC Web Service Developers" on page 2-1
- "Developing JAX-RPC Web Services" on page 3-1
- "Programming the JWS File" on page 4-1
- "Developing JAX-RPC Web Service Clients" on page 6-1

Table 10–1 Steps to Create a Buffered Web Service

#	Step	Description
1	Configure the WebLogic Server instance that hosts the buffered Web service.	See Section 10.3, "Configuring the Host WebLogic Server Instance for the Buffered Web Service".
2	Create a new JWS file, or update an existing one, that implements the buffered Web service.	Use your favorite IDE or text editor. See Section 10.4, "Programming Guidelines for the Buffered JWS File".

3	Update your `build.xml` file to include a call to the `jwsc` Ant task to compile the JWS file into a buffered Web service.	For example:
		`<jwsc`
		`srcdir="src"`
		`destdir="${service-ear-dir}" >`
		`<jws`
		`file="examples/webservices/async_buffered/AsyncBufferedImpl.java"`
		`/>`
		`</jwsc>`
		
		See "Running the jwsc WebLogic Web Services Ant Task" on page 3-7 for general information about using the `jwsc` Ant task.

4	Recompile your JWS file by calling the appropriate target, then redeploy the Web service to the WebLogic Server.	For example: `prompt> ant build-clientService deploy-clientService` For more information about deployment, see "Deploying and Undeploying WebLogic Web Services" on page 3-14.
5	Create a new JWS file, or update an existing one, that implements the client Web service that invokes the buffered Web service.	See Section 10.5, "Programming the JWS File That Invokes the Buffered Web Service".
6	Update the `build.xml` file that builds the client Web service.	See Section 10.6, "Updating the build.xml File for a Client of the Buffered Web Service".
7	Recompile your client JWS file by calling the appropriate target, then redeploy the Web service to the client WebLogic Server.	For example: `prompt> ant build-clientService deploy-clientService` For more information about deployment, see "Deploying and Undeploying WebLogic Web Services" on page 3-14.

10.3 Configuring the Host WebLogic Server Instance for the Buffered Web Service

Configuring the WebLogic Server instance on which the buffered Web service is deployed involves configuring JMS resources, such as JMS servers and modules, that are used internally by the Web services runtime.

You can configure these resources manually or you can use the Configuration Wizard to extend the WebLogic Server domain using a Web services-specific extension template. Using the Configuration Wizard greatly simplifies the required configuration steps; for details, see "Configuring Your Domain For Web Services Features" on page 3-2.

Notes: Alternatively, you can use WLST to configure the resources. For information about using WLST to extend the domain, see "Configuring Existing Domains" in *Understanding the WebLogic Scripting Tool*.

A domain that does not contain Web Services resources will still boot and operate correctly for non-Web services scenarios, and any Web Services scenario that does not involve asynchronous request and response. You will, however, see INFO messages in the server log indicating that asynchronous resources have not been configured and that the asynchronous response service for Web services has not been completely deployed.

If you prefer to configure the resources manually, perform the following steps.

Table 10–2 Steps to Configure Host WebLogic Server Instance Manually for the Buffered Web Service

#	Step	Description
1	Invoke the Administration Console for the domain that contains the host WebLogic Server instance.	To invoke the Administration Console in your browser, enter the following URL: `http://host:port/console` where ▪ *`host`* refers to the computer on which the Administration Server is running. ▪ *`port`* refers to the port number where the Administration Server is listening for connection requests. The default port number for the Administration server is 7001. See "Invoking the Administration Console" in *Understanding WebLogic Web Services for Oracle WebLogic Server*.
3	Create a JMS Server.	Create a JMS Server. If a JMS server already exists, you can use it if you do not want to create a new one. See "Create JMS servers" in *Oracle WebLogic Server Administration Console Online Help*.

Table 10–2 (Cont.) Steps to Configure Host WebLogic Server Instance Manually for the Buffered Web

#	Step	Description
4	Create JMS module and define queue.	Create a JMS module, and then define a JMS queue in the module. If a JMS module already exists, you can use it if you do not want to create a new one. Target the JMS queue to the JMS server you created in the preceding step. Be sure you specify that this JMS queue is local, typically by setting the local JNDI name. See "Create JMS system modules" and "Create queues in a system module" in *Oracle WebLogic Server Administration Console Online Help*. If you want the buffered Web service to use the default Web services queue, set the JNDI name of the JMS queue to `weblogic.wsee.DefaultCallbackQueue`. Otherwise, if you use a different JNDI name, be sure to use the `@BufferQueue` annotation in the JWS file to specify this JNDI name to the reliable Web service. See Section 10.4, "Programming Guidelines for the Buffered JWS File". **Clustering Considerations:** If you are using the Web service buffering feature in a cluster, you must: ■ Create a *local* JMS queue, rather than a distributed queue, when creating the JMS queue. ■ Explicitly target this JMS queue to each server in the cluster.
4	Tune your domain environment, as required. (Optional)	Review "Tuning Heavily Loaded Systems to Improve Web Service Performance" in *WebLogic Server Performance and Tuning*.

10.4 Programming Guidelines for the Buffered JWS File

The following example shows a simple JWS file that implements a buffered Web service; see the explanation after the example for coding guidelines that correspond to the Java code in bold.

```
package examples.webservices.buffered;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.Oneway;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.MessageBuffer;
import weblogic.jws.BufferQueue;

@WebService(name="BufferedPortType",
            serviceName="BufferedService",
            targetNamespace="http://example.org")

@WLHttpTransport(contextPath="buffered",
                 serviceUri="BufferedService",
                 portName="BufferedPort")

// Annotation to specify a specific JMS queue rather than the default
@BufferQueue(name="my.jms.queue")

/**
 * Simple buffered Web Service.
 */

public class BufferedImpl {

  @WebMethod()
```

```
  @MessageBuffer(retryCount=10, retryDelay="10 seconds")
  @Oneway()
  public void sayHelloNoReturn(String message) {
    System.out.println("sayHelloNoReturn: " + message);

  }
}
```

Follow these guidelines when programming the JWS file that implements a buffered Web service. Code snippets of the guidelines are shown in bold in the preceding example.

- Import the JWS annotations used for buffered Web services:

  ```
  import javax.jws.Oneway;
  import weblogic.jws.MessageBuffer;
  import weblogic.jws.BufferQueue;
  ```

  See the following bullets for guidelines on which JWS annotations are required.

- Optionally use the class-level `@BufferQueue` JWS annotation to specify the JNDI name of the JMS queue used internally by WebLogic Server when it processes a buffered invoke; for example:

  ```
  @BufferQueue(name="my.jms.queue")
  ```

  If you do not specify this JWS annotation, then WebLogic Server uses the default Web services JMS queue (`weblogic.wsee.DefaultQueue`).

  You must create both the default JMS queue and any queues specified with this annotation before you can successfully invoke a buffered operation. See Section 10.3, "Configuring the Host WebLogic Server Instance for the Buffered Web Service" for details.

- Use the `@MessageBuffer` JWS annotation to specify the operations of the Web service that are buffered. The annotation has two optional attributes:

  - `retryCount`: The number of times WebLogic Server should attempt to deliver the message from the JMS queue to the Web service implementation (default 3).
  - `retryDelay`: The amount of time that the server should wait in between retries (default 5 minutes).

  For example:

  ```
  @MessageBuffer(retryCount=10, retryDelay="10 seconds")
  ```

  You can use this annotation at the class-level to specify that all operations are buffered, or at the method-level to choose which operations are buffered.

- If you plan on invoking the buffered Web service operation synchronously (or in other words, *not* using the asynchronous request-response feature), then the implementing method is required to be annotated with the `@Oneway` annotation to specify that the method is one-way. This means that the method cannot return a value, but rather, must explicitly return `void`. For example:

  ```
  @Oneway()
  public void sayHelloNoReturn(String message) {
  ```

  Conversely, if the method is *not* annotated with the `@Oneway` annotation, then you must invoke it using the asynchronous request-response feature. If you are unsure

how the operation is going to be invoked, consider creating two flavors of the operation: synchronous and asynchronous.

See Chapter 7, "Invoking a Web Service Using Asynchronous Request-Response," and Chapter 11, "Using the Asynchronous Features Together."

10.5 Programming the JWS File That Invokes the Buffered Web Service

You can invoke a buffered Web service from both a stand-alone Java application (if not using asynchronous request-response) and from another Web service. Unlike other WebLogic Web services asynchronous features, however, you do not use the `@ServiceClient` JWS annotation in the client Web service, but rather, you invoke the service as you would any other. For details, see "Invoking a Web Service from Another Web Service" on page 6-9.

The following sample JWS file shows how to invoke the `sayHelloNoReturn` operation of the `BufferedService` Web service:

```
package examples.webservices.buffered;

import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;

import javax.jws.WebService;
import javax.jws.WebMethod;

import weblogic.jws.WLHttpTransport;

import examples.webservices.buffered.BufferedPortType;
import examples.webservices.buffered.BufferedService_Impl;
import examples.webservices.buffered.BufferedService;

@WebService(name="BufferedClientPortType",
            serviceName="BufferedClientService",
            targetNamespace="http://examples.org")

@WLHttpTransport(contextPath="bufferedClient",
                 serviceUri="BufferedClientService",
                 portName="BufferedClientPort")

public class BufferedClientImpl {

  @WebMethod()
  public String callBufferedService(String input, String serviceUrl)
        throws RemoteException {

    try {

    BufferedService service = new BufferedService_Impl(serviceUrl + "?WSDL");
    BufferedPortType port = service.getBufferedPort();

    // Invoke the sayHelloNoReturn() operation of BufferedService

    port.sayHelloNoReturn(input);

    return "Invoke went okay!";

    } catch (ServiceException se) {

         System.out.println("ServiceExcpetion thrown");
```

```
            throw new RuntimeException(se);

        }
    }
}
```

10.6 Updating the build.xml File for a Client of the Buffered Web Service

To update a `build.xml` file to generate the JWS file that invokes a buffered Web service operation, add `taskdefs` and a `build-clientService` targets that look something like the following example. See the description after the example for details.

```
<taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JwscTask" />

<target name="build-clientService">

    <jwsc
        enableAsyncService="true"
        srcdir="src"
        destdir="${clientService-ear-dir}" >

        <jws file="examples/webservices/buffered/BufferedClientImpl.java">
           <clientgen

wsdl="http://${wls.hostname}:${wls.port}/buffered/BufferedService?WSDL"
             packageName="examples.webservices.buffered"/>

        </jws>

    </jwsc>

  </target>
```

Use the `taskdef` Ant task to define the full classname of the `jwsc` Ant tasks.

Update the `jwsc` Ant task that compiles the client Web service to include a `<clientgen>` child element of the `<jws>` element so as to generate and compile the JAX-RPC stubs for the deployed `BufferedService` Web service. The `jwsc` Ant task automatically packages them in the generated WAR file so that the client Web service can immediately access the stubs. You do this because the `BufferedClientImpl` JWS file imports and uses one of the generated classes.

11

Using the Asynchronous Features Together

This chapter describes how to use the asynchronous features together with WebLogic Java API for XML-based RPC (JAX-RPC) Web services.

This chapter includes the following topics:

- Section 11.1, "Using the Asynchronous Features Together"
- Section 11.2, "Example of a JWS File That Implements a Reliable Conversational Web Service"
- Section 11.3, "Example of Client Web Service That Asynchronously Invokes a Reliable Conversational Web Service"

11.1 Using the Asynchronous Features Together

The preceding sections describe how to use the WebLogic Web service asynchronous features (Web service reliable messaging, conversations, asynchronous request-response, and buffering) on their own. Typically, however, Web services use the features together; see Section 11.2, "Example of a JWS File That Implements a Reliable Conversational Web Service" and Section 11.3, "Example of Client Web Service That Asynchronously Invokes a Reliable Conversational Web Service" for examples.

When used together, some restrictions described in the individual feature sections do not apply, and sometimes additional restrictions apply. The following table summarizes considerations for various feature combinations.

Table 11–1 Considerations When Using Asynchronous Features Together

Feature Combination	Consideration
Asynchronous request-response with Web service reliable messaging or buffering	▪ The asynchronous response from the reliable Web service is also reliable. This means that you must also configure a JMS server, module, and queue on the *source* WebLogic Server instance, in a similar way you configured the destination WebLogic Server instance, to handle the response. When you create the JMS queue on the source WebLogic Server instance, you are required to specify a JNDI name of `weblogic.wsee.DefaultQueue`; you can name the queue anything you want. You must also ensure that you specify that this JMS queue is *local*, typically by setting the local JNDI name. ▪ The reliable or buffered operation *cannot* be one-way; in other words, you cannot annotate the implementing method with the `@Oneway` annotation.
Asynchronous request-response with Web service reliable messaging	If you set a property in one of the asynchronous contexts (`AsyncPreCallContext` or `AsyncPostCallContext`), then the property must implement `java.io.Serializable`.
Asynchronous request-response with buffering	You must use the `@ServiceClient` JWS annotation in the client Web service that invokes the buffered Web service operation.
Conversations with Web service reliable messaging	▪ JWS conversations are *not* the same as reliable sequences, and are not linked in any way. You must consider the management of reliable sequences separately from the life cycle of a conversation. For example, when using reliable messaging to send messages between a client service and a reliable and conversational service, finishing the conversation does not terminate the reliable sequence. You must explicitly cause the reliable sequence to be terminated (using `WsrmUtils.setFinalMessage()` or other acceptable method) or allows the reliable sequence to remain active until it expires when the sequence lifetime is exceeded). For more information about reliable message sequence life cycle, see Section 8.1.2, "Managing the Life Cycle of the Reliable Message Sequence". ▪ If you set the property `WLStub.CONVERSATIONAL_METHOD_BLOCK_TIMEOUT` on the stub of the client Web service, the property is ignored because the client does not block. ▪ At least one method of the reliable conversational Web service must *not* be marked with the `@Oneway` annotation.
Conversations with asynchronous request-response	Asynchronous responses between a client conversational Web service and any other Web service also participate in the conversation. For example, assume WebServiceA is conversational, and it invokes WebServiceB using asynchronous request-response. Because WebServiceA is conversational the asynchronous responses from WebServiceB also participates in the same conversation.

11.2 Example of a JWS File That Implements a Reliable Conversational Web Service

The following sample JWS file implements a Web service that is both reliable and conversational:

```
package examples.webservices.async_mega;

import java.io.Serializable;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.Conversation;
import weblogic.jws.Policy;

import javax.jws.WebService;
```

```
import javax.jws.WebMethod;

@WebService(name="AsyncMegaPortType",
            serviceName="AsyncMegaService",
            targetNamespace="http://examples.org/")

@Policy(uri="AsyncReliableConversationPolicy.xml",
        attachToWsdl=true)

@WLHttpTransport(contextPath="asyncMega",
                 serviceUri="AsyncMegaService",
                 portName="AsyncMegaServicePort")

/**
 * Web Service that is both reliable and conversational.
 */

public class AsyncMegaServiceImpl implements Serializable {

  @WebMethod
  @Conversation (Conversation.Phase.START)
  public String start() {
    return "Starting conversation";
  }

  @WebMethod
  @Conversation (Conversation.Phase.CONTINUE)
  public String middle(String message) {
    return "Middle of conversation; the message is: " + message;
  }

  @WebMethod
  @Conversation (Conversation.Phase.FINISH)
  public String finish(String message ) {
    return "End of conversation; the message is: " + message;
  }

}
```

11.3 Example of Client Web Service That Asynchronously Invokes a Reliable Conversational Web Service

The following JWS file shows how to implement a client Web service that reliably invokes the various conversational methods of the Web service described in Section 11.2, "Example of a JWS File That Implements a Reliable Conversational Web Service"; the client JWS file uses the asynchronous request-response feature as well.

```
package examples.webservices.async_mega;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;
import weblogic.jws.AsyncResponse;
import weblogic.jws.AsyncFailure;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.xml.rpc.Stub;

import weblogic.wsee.async.AsyncPreCallContext;
```

```
import weblogic.wsee.async.AsyncCallContextFactory;
import weblogic.wsee.async.AsyncPostCallContext;
import weblogic.wsee.reliability.WsrmUtils;

import examples.webservices.async_mega.AsyncMegaPortType;
import examples.webservices.async_mega.AsyncMegaService;
import examples.webservices.async_mega.AsyncMegaService_Impl;

import java.rmi.RemoteException;

@WebService(name="AsyncMegaClientPortType",
            serviceName="AsyncMegaClientService",
            targetNamespace="http://examples.org/")

@WLHttpTransport(contextPath="asyncMegaClient",
                 serviceUri="AsyncMegaClient",
                 portName="AsyncMegaClientServicePort")

/**
 * Client Web Service that has a conversation with the AsyncMegaService
 * reliably and asynchronously.
 */

public class AsyncMegaClientImpl {

  @ServiceClient(
     wsdlLocation="http://localhost:7001/asyncMega/AsyncMegaService?WSDL",
     serviceName="AsyncMegaService",
     portName="AsyncMegaServicePort")

  private AsyncMegaPortType port;

  @WebMethod
  public void runAsyncReliableConversation(String message) {

    AsyncPreCallContext apc = AsyncCallContextFactory.getAsyncPreCallContext();
    apc.setProperty("message", message);

    try {
      port.startAsync(apc);
      System.out.println("start method executed.");

      port.middleAsync(apc, message );
      System.out.println("middle method executed.");

      // Since this service is not conversational, any state kept in the port
      // field will be lost when this method returns. In the case of reliable
      // messaging, this state includes the ID of the reliable sequence being
      // used to send messages. The setFinalMessage method specifies
      // that this is the final message to be sent on this sequence. This
      // will allow the reliable messaging subsystem to proactively clean up
      // the reliable sequence instead of timing out.
      WsrmUtils.setFinalMessage((Stub)port);
      port.finishAsync(apc, message );
      System.out.println("finish method executed.");

    }
    catch (RemoteException e) {
      e.printStackTrace();
    }
```

```
  }

  @AsyncResponse(target="port", operation="start")
  public void onStartAsyncResponse(AsyncPostCallContext apc, String message) {
    System.out.println("--------------------");
    System.out.println("Got message " + message );
    System.out.println("--------------------");
  }

  @AsyncResponse(target="port", operation="middle")
  public void onMiddleAsyncResponse(AsyncPostCallContext apc, String message) {
    System.out.println("--------------------");
    System.out.println("Got message " + message );
    System.out.println("--------------------");
  }

  @AsyncResponse(target="port", operation="finish")
  public void onFinishAsyncResponse(AsyncPostCallContext apc, String message) {
    System.out.println("--------------------");
    System.out.println("Got message " + message );
    System.out.println("--------------------");
  }

  @AsyncFailure(target="port", operation="start")
  public void onStartAsyncFailure(AsyncPostCallContext apc, Throwable e) {
    System.out.println("--------------------");
    e.printStackTrace();
    System.out.println("--------------------");
  }

  @AsyncFailure(target="port", operation="middle")
  public void onMiddleAsyncFailure(AsyncPostCallContext apc, Throwable e) {
    System.out.println("--------------------");
    e.printStackTrace();
    System.out.println("--------------------");
  }

  @AsyncFailure(target="port", operation="finish")
  public void onFinishAsyncFailure(AsyncPostCallContext apc, Throwable e) {
    System.out.println("--------------------");
    e.printStackTrace();
    System.out.println("--------------------");
  }
}
```

12

Using Callbacks to Notify Clients of Events

This chapter describes how to callbacks with WebLogic Java API for XML-based RPC (JAX-RPC) Web services to notify clients of events.

This chapter includes the following topics:

12.1 Overview of Callbacks

Callbacks notify a client of your Web service that some event has occurred. For example, you can notify a client when the results of that client's request are ready, or when the client's request cannot be fulfilled.

When you expose a method as a standard public operation in your JWS file (by using the `@WebMethod` annotation), the client sends a SOAP message to the Web service to invoke the operation. When you add a callback to a Web service, however, you define a message that the Web service sends *back to the client Web service*, notifying the client of an event that has occurred. So exposing a method as a public operation and defining a callback are completely symmetrical processes, with opposite recipients.

WebLogic Server automatically routes the SOAP message from client invoke to the target Web service. In order to receive callbacks, however, the client must be operating in an environment that provides the same services. This typically means the client is a Web service running on a Web server. If the client does not meet these requirements, it is likely not capable of receiving callbacks from your Web service.

The protocol and message format used for callbacks is always the same as the protocol and message format used by the conversation start method that initiated the current conversation. If you attempt to override the protocol or message format of a callback, an error is thrown.

12.2 Callback Implementation Overview and Terminology

To implement callbacks, you must create or update the following three Java files:

- *Callback interface*: Java interface file that defines the callback methods. You do not explicitly implement this file yourself; rather, the `jwsc` Ant task automatically generates an implementation of the interface. The implementation simply passes a message from the target Web service back to the client Web service. The generated Web service is deployed to the same WebLogic Server that hosts the client Web service.

  In the example in this section, the callback interface is called `CallbackInterface`. The interface defines a single callback method called `callbackOperation()`.

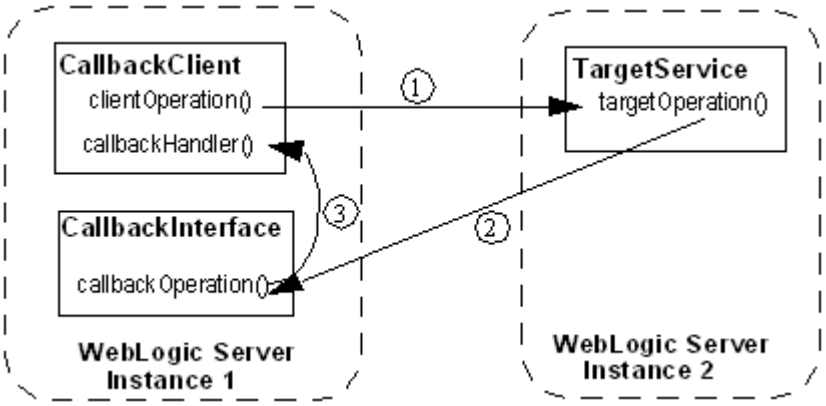
- **JWS file that implements the *target Web service***: The target Web service includes one or more standard operations that invoke a method defined in the callback interface; this method in turn sends a message back to the client Web service that originally invoked the operation of the target Web service.

  In the example, this Web service is called `TargetService` and it defines a single standard method called `targetOperation()`.

- **JWS file that implements the *client Web service***: The client Web service invokes an operation of the target Web service. This Web service includes one or more methods that specify what the client should do when it receives a callback message back from the target Web service via a callback method.

  In the example, this Web service is called `CallbackClient` and the method that is automatically invoked when it receives a callback is called `callbackHandler()`. The method that invokes `TargetService` in the standard way is called `clientOperation()`.

The following graphic shows the flow of messages:



1. The `clientOperation()` method of the `CallbackClient` Web service, running in one WebLogic Server instance, explicitly invokes the `targetOperation()` operation of the `TargetService`. The `TargetService` service might be running in a separate WebLogic Server instance.
2. The implementation of the `TargetService.targetOperation()` method explicitly invokes the `callbackOperation()` operation of the `CallbackInterface`, which implements the callback service. The callback service is deployed to the WebLogic Server which hosts the client Web service.
3. The `jwsc`-generated implementation of the `CallbackInterface.callbackOperation()` method simply sends a message back to the `CallbackClient` Web service. The client Web service includes a method `callbackHandler()` that handles this message.

12.3 Programming Callbacks: Main Steps

The procedure in this section describes how to program and compile the three JWS files that are required to implement callbacks: the target Web service, the client Web service, and the callback interface. The procedure shows how to create the JWS files from scratch; if you want to update existing JWS files, you can also use this procedure as a guide.

It is assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file to which you can add targets for running the `jwsc` Ant task and deploying the Web services.

Table 12–1 Steps to Program Callbacks

#	Step	Description
1	Create a new JWS file, or update an existing one, that implements the target Web service.	Use your favorite IDE or text editor. See Section 12.4, "Programming Guidelines for Target Web Service". **Note:** The JWS file that implements the target Web service invokes one or more callback methods of the callback interface. However, the step that describes how to program the callback interface comes later in this procedure. For this reason, programmers typically program the three JWS files at the same time, rather than linearly as implied by this procedure. The steps are listed in this order for clarity only.
2	Update your `build.xml` file to include a call to the `jwsc` Ant task to compile the target JWS file into a Web service.	See "Running the jwsc WebLogic Web Services Ant Task" on page 3-7.
3	Run the Ant target to build the target Web service.	For example: `prompt> ant build-mainService`
4	Deploy the target Web service as usual.	See "Deploying and Undeploying WebLogic Web Services" on page 3-14.
5	Create a new JWS file, or update an existing one, that implements the client Web service.	It is assumed that the client Web service is deployed to a different WebLogic Server instance from the one that hosts the target Web service. See Section 12.5, "Programming Guidelines for the Callback Client Web Service".
6	Create the callback JWS interface that implements the callback Web service.	See Section 12.6, "Programming Guidelines for the Callback Interface".
7	Update the `build.xml` file that builds the client Web service.	The `jwsc` Ant task that builds the client Web service also implicitly generates the callback Web service from the callback interface file. See Section 12.7, "Updating the build.xml File for the Client Web Service".
8	Run the Ant target to build the client and callback Web services.	For example: `prompt> ant build-clientService`
9	Deploy the client Web service as usual.	See "Deploying and Undeploying WebLogic Web Services" on page 3-14.

12.4 Programming Guidelines for Target Web Service

The following example shows a simple JWS file that implements the target Web service; see the explanation after the example for coding guidelines that correspond to the Java code in **bold**.

```
package examples.webservices.callback;

import weblogic.jws.WLHttpTransport;
```

```
import weblogic.jws.Callback;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService(name="CallbackPortType",
            serviceName="TargetService",
            targetNamespace="http://examples.org/")

@WLHttpTransport(contextPath="callback",
                 serviceUri="TargetService",
                 portName="TargetServicePort")

/**
 * callback service
 */

public class TargetServiceImpl {

  @Callback
  CallbackInterface callback;

  @WebMethod
  public void targetOperation (String message) {

        callback.callbackOperation (message);
  }

}
```

Follow these guidelines when programming the JWS file that implements the target Web service. Code snippets of the guidelines are shown in bold in the preceding example.

- Import the required JWS annotations:

  ```
  import weblogic.jws.Callback;
  ```

- Use the `@weblogic.jws.Callback` JWS annotation to specify that a variable is a callback, which means that you can use the annotated variable to send callback events back to a client Web service that invokes an operation of the `TargetService` Web service. The data type of the variable is the callback interface, which in this case is called `CallbackInterface`.

  ```
  @Callback
  CallbackInterface callback;
  ```

- In a method that implements an operation of the `TargetService`, use the annotated variable to invoke one of the callback methods of the callback interface, which in this case is called `callbackOperation()`:

  ```
  callback.callbackOperation (message);
  ```

See "JWS Annotation Reference" in *WebLogic Web Services Reference for Oracle WebLogic Server* for additional information about the WebLogic-specific JWS annotations discussed in this section.

12.5 Programming Guidelines for the Callback Client Web Service

The following example shows a simple JWS file for a client Web service that invokes the target Web service described in Section 12.4, "Programming Guidelines for Target Web Service"; see the explanation after the example for coding guidelines that correspond to the Java code in **bold**.

```
package examples.webservices.callback;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;
import weblogic.jws.CallbackMethod;
import weblogic.jws.security.CallbackRolesAllowed;
import weblogic.jws.security.SecurityRole;

import javax.jws.WebService;
import javax.jws.WebMethod;

import examples.webservices.callback.CallbackPortType;

import java.rmi.RemoteException;

@WebService(name="CallbackClientPortType",
            serviceName="CallbackClientService",
            targetNamespace="http://examples.org/")

@WLHttpTransport(contextPath="callbackClient",
                 serviceUri="CallbackClient",
                 portName="CallbackClientPort")

public class CallbackClientImpl {

  @ServiceClient(
     wsdlLocation="http://localhost:7001/callback/TargetService?WSDL",
     serviceName="TargetService",
     portName="TargetServicePort")
  @CallbackRolesAllowed(@SecurityRole(role="mgr", mapToPrincipals="joe"))
  private CallbackPortType port;

  @WebMethod
  public void clientOperation (String message) {

    try {

        port.targetOperation(message);
    }
    catch (RemoteException e) {
      e.printStackTrace();
    }

  }
  @CallbackMethod(target="port", operation="callbackOperation")
  @CallbackRolesAllowed(@SecurityRole(role="engineer",
mapToPrincipals="shackell"))
  public void callbackHandler(String msg) {

        System.out.println (msg);
  }

}
```

Follow these guidelines when programming the JWS file that invokes the target Web service; code snippets of the guidelines are shown in bold in the preceding example:

- Import the required JWS annotations:

```
import weblogic.jws.ServiceClient;
import weblogic.jws.CallbackMethod;
```

- Optionally import the security-related annotations if you want to specify the roles that are allowed to invoke the callback methods:

```
import weblogic.jws.security.CallbackRolesAllowed;
import weblogic.jws.security.SecurityRole;
```

- Import the JAX-RPC stub of the port type of the target Web service you want to invoke. The actual stub itself will be created later by the `jwsc` Ant task. The stub package is specified by the `packageName` attribute of the `<clientgen>` child element of `<jws>`, and the name of the stub is determined by the WSDL of the invoked Web service.

```
import examples.webservices.callback.CallbackPortType;
```

- In the body of the JWS file, use the `@ServiceClient` JWS annotation to specify the WSDL, name, and port of the target Web service you want to invoke. You specify this annotation at the field-level on a private variable, whose data type is the JAX-RPC port type of the Web service you are invoking.

```
@ServiceClient(
   wsdlLocation="http://localhost:7001/callback/TargetService?WSDL",
   serviceName="TargetService",
   portName="TargetServicePort")
@CallbackRolesAllowed(@SecurityRole(role="mgr", mapToPrincipals="joe"))
private CallbackPortType port;
```

  The preceding code also shows how to use the optional `@CallbackRolesAllowed` annotation to specify the list of `@SecurityRoles` that are allowed to invoke the callback methods.

- Using the variable you annotated with the `@ServiceClient` annotation, invoke an operation of the target Web service. This operation in turn will invoke a callback method of the callback interface:

```
port.targetOperation(message);
```

- Create a method that will handle the callback message received from the callback service. You can name this method anything you want. However, its signature should exactly match the signature of the corresponding method in the callback interface.

  Annotate the method with the `@CallbackMethod` annotation to specify that this method handles callback messages. Use the `target` attribute to specify the name of the JAX-RPC port for which you want to receive callbacks (in other words, the variable you previously annotated with `@ServiceClient`). Use the `operation` attribute to specify the name of the callback method in the callback interface from which this method will handle callback messages.

```
  @CallbackMethod(target="port", operation="callbackOperation")
  @CallbackRolesAllowed(@SecurityRole(role="engineer",
mapToPrincipals="shackell"))
  public void callbackHandler(String msg) {
        System.out.println (msg);
```

```
}
```

The preceding code also shows how to use the optional `@CallbackRolesAllowed` annotation to further restrict the security roles that are allowed to invoke this particular callback method.

See "JWS Annotation Reference" in *WebLogic Web Services Reference for Oracle WebLogic Server* for additional information about the WebLogic-specific JWS annotations discussed in this section.

12.6 Programming Guidelines for the Callback Interface

The callback interface is also a JWS file that implements a Web service, except for one big difference: instead of using the standard `@javax.jws.WebService` annotation to specify that it is a standard Web service, you use the WebLogic-specific `@weblogic.jws.CallbackService` to specify that it is a callback service. The attributes of `@CallbackService` are a restricted subset of the attributes of `@WebService`.

Follow these restrictions on the allowed data types and JWS annotations when programming the JWS file that implements a callback service:

- You cannot use any WebLogic-specific JWS annotations other than `@weblogic.jws.CallbackService`.
- You can use all standard JWS annotations except for the following:
  - `javax.jws.HandlerChain`
  - `javax.jws.soap.SOAPMessageHandler`
  - `javax.jws.soap.SOAPMessageHandlers`
- You can use all supported data types as parameters or return values except `Holder` classes (user-defined data types that implement the `javax.xml.rpc.holders.Holder` interface).

The following example shows a simple callback interface file that implements a callback Web service. The target Web service, described in Section 12.4, "Programming Guidelines for Target Web Service", explicitly invokes a method in this interface. The `jwsc`-generated implementation of the callback interface then automatically sends a message back to the client Web service that originally invoked the target Web service; the client service is described in Section 12.5, "Programming Guidelines for the Callback Client Web Service". See the explanation after the example for coding guidelines that correspond to the Java code in **bold**.

```
package examples.webservices.callback;

import weblogic.jws.CallbackService;

import javax.jws.Oneway;
import javax.jws.WebMethod;

@CallbackService
public interface CallbackInterface {

  @WebMethod
  @Oneway
  public void callbackOperation (String msg);

}
```

Follow these guidelines when programming the JWS interface file that implements the callback Web service. Code snippets of the guidelines are shown in **bold** in the preceding example.

- Import the required JWS annotation:

  ```
  import weblogic.jws.CallbackService;
  ```

- Annotate the interface declaration with the `@CallbackService` annotation to specify that the JWS file implements a callback service:

  ```
  @CallbackService
  public interface CallbackInterface {
  ```

- Create a method that the target Web service explicitly invokes; this is the method that automatically sends a message back to the client service that originally invoked the target Web service. Because this is a Java interface file, you do not provide an implementation of this method. Rather, the WebLogic Web services runtime generates an implementation of the method via the `jwsc` Ant task.

  ```
  public void callbackOperation (String msg);
  ```

  Note: Although the example shows the callback method returning void and annotated with the `@Oneway` annotation, this is not a requirement.

See "JWS Annotation Reference" in *WebLogic Web Services Reference for Oracle WebLogic Server* for additional information about the WebLogic-specific JWS annotations discussed in this section.

12.7 Updating the build.xml File for the Client Web Service

When you run the `jwsc` Ant task against the JWS file that implements the client Web service, the task implicitly also generates the callback Web service, as described in this section.

You update a `build.xml` file to generate a client Web service that invokes the target Web service by adding `taskdefs` and a `build-clientService` target that looks something like the following example. See the description after the example for details.

```
<taskdef name="jwsc"
  classname="weblogic.wsee.tools.anttasks.JwscTask" />

<target name="build-clientService">

  <jwsc
      srcdir="src"
      destdir="${clientService-ear-dir}" >

      <jws file="examples/webservices/callback/CallbackClientImpl.java" >

        <clientgen
          wsdl="http://${wls.hostname}:${wls.port}/callback/TargetService?WSDL"
          packageName="examples.webservices.callback"
          serviceName="TargetService" />

      </jws>
```

```
  </jwsc>

</target>
```

Use the `taskdef` Ant task to define the full classname of the `jwsc` Ant tasks.

Update the `jwsc` Ant task that compiles the client Web service to include a `<clientgen>` child element of the `<jws>` element so as to generate and compile the JAX-RPC stubs for the deployed `TargetService` Web service. The `jwsc` Ant task automatically packages them in the generated WAR file so that the client Web service can immediately access the stubs. You do this because the `CallbackClientImpl` JWS file imports and uses one of the generated classes.

Because the WSDL of the target Web service includes an additional `<service>` element that describes the callback Web service (which the target Web service invokes), the `<clientgen>` child element of the `jwsc` Ant task also generates and compiles the callback Web service and packages it in the same EAR file as the client Web service.

13

Using JMS Transport as the Connection Protocol

This chapter describes how to use JMS transport as the connection protocol with WebLogic Java API for XML-based RPC (JAX-RPC) Web service using asynchronous request-response.

This chapter includes the following topics:

13.1 Overview of Using JMS Transport

Typically, client applications use HTTP/S as the connection protocol when invoking a WebLogic Web service. You can, however, configure a WebLogic Web service so that client applications use JMS as the transport instead.

Using JMS transport offers the following benefits: reliability, scalability, and quality of service. As with Web service reliable messaging, if WebLogic Server goes down while the method invocation is still in the queue, it will be dealt with as soon as WebLogic Server is restarted. When a client invokes a Web service, the client does not wait for a response from the invoke, and the execution of the client can continue. Using JMS transport does require slightly more overhead and programming complexity than HTTP/S.

You configure transports using either JWS annotations or child elements of the `jwsc` Ant task, as described in later sections. When a WebLogic Web service is configured to use JMS as the connection transport, the endpoint address specified for the corresponding port in the generated WSDL of the Web service uses `jms://` in its URL rather than `http://`. An example of a JMS endpoint address is as follows:

```
jms://myHost:7001/transports/JMSTransport?URI=JMSTransportQueue
```

The `URI=JMSTransportQueue` section of the URL specifies the JMS queue that has been configured for the JMS transport feature. Although you cannot invoke the Web service using HTTP, you can view its WSDL using HTTP, which is how the `clientgen` is still able to generate JAX-RPC stubs for the Web service.

For each transport that you specify, WebLogic Server generates an additional port in the WSDL. For this reason, if you want to give client applications a choice of transports they can use when they invoke the Web service (JMS, HTTP, or HTTPS), you should explicitly add the transports using the appropriate JWS annotations or child elements of `jwsc`.

Note: Using JMS transport is an added-value WebLogic feature; non-WebLogic client applications, such as a .NET client, may not be able to invoke the Web service using the JMS port.

13.2 Using JMS Transport Starting From Java: Main Steps

To use JMS transport when starting from Java, you must perform at least one of the following tasks:

- Add the `@WLJmsTransport` annotation to your JWS file.
- Add a `<WLJmsTransport>` child element to the `jwsc` Ant task. This setting overrides the transports defined in the JWS file.

Note: Because you might not know at the time that you are coding the JWS file which transport best suits your needs, it is often better to specify the transport at build-time using the `<WLJmsTransport>` child element.

The following procedure describes the complete set of steps required so that your Web service can be invoked using the JMS transport when starting from Java.

Note: It is assumed that you have created a basic JWS file that implements a Web service and that you want to configure the Web service to be invoked using JMS. It is also assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file that includes targets for running the `jwsc` Ant task and deploying the service.

Table 13–1 Steps to Use JMS Transport Starting From Java

#	Step	Description
1	Configure the WebLogic Server domain for the required JMS components.	See Section 13.4, "Configuring the Host WebLogic Server Instance for the JMS Transport Web Service".
2	Add the `@WLJmsTransport` annotation to your JWS file. (Optional)	This step is optional. If you do not add the `@WLJmsTransport` annotation to your JWS file, then you must add a `<WLJmsTransport>` child element to the `jwsc` Ant task, as described in Step 3. See Section 13.5, "Using the @WLJmsTransport JWS Annotation".

Table 13–1 (Cont.) Steps to Use JMS Transport Starting From Java

#	Step	Description
3	Add a `<WLJmsTransport>` child element to the `jwsc` Ant task. (Optional)	Use the `<WLJmsTransport>` child element to override the transports defined in the JWS file. This step is required if you did not add the `@WLJmsTransport` annotation to your JWS file in Step 2. Otherwise, this step is optional. See Section 13.6, "Using the <WLJmsTransport> Child Element of the jwsc Ant Task" for details.
4	Build your Web service by running the target in the `build.xml` Ant file that calls the `jwsc` task.	For example, if the target that calls the `jwsc` Ant task is called `build-service`, then you would run: `prompt> ant build-service` See "Running the jwsc WebLogic Web Services Ant Task" on page 3-7.
5	Deploy your Web service to WebLogic Server.	See "Deploying and Undeploying WebLogic Web Services" on page 3-14.

See Section 13.8, "Invoking a WebLogic Web Service Using JMS Transport" for information about updating your client application to invoke the Web service using JMS transport.

13.3 Using JMS Transport Starting From WSDL: Main Steps

To use JMS transport when starting from WSDL, you must perform at least one of the following tasks:

- Update the WSDL to use JMS transport before running the `wsdlc` Ant task.
- Update the stubbed-out JWS implementation file generated by the `wsdlc` Ant task to add the `@WLJmsTransport` annotation.
- Add a `<WLJmsTransport>` child element to the `jwsc` Ant task used to build the JWS implementation file. This setting overrides the transports defined in the JWS file.

> **Note:** Because you might not know at the time that you are coding the JWS file which transport best suits your needs, it is often better to specify the transport at build-time using the `<WLJmsTransport>` child element.

The following procedure describes the complete set of steps required so that your Web service can be invoked using the JMS transport when starting from WSDL.

> **Note:** It is assumed in this procedure that you have an existing WSDL file.

Table 13–2 Steps to Use JMS Transport Starting From WSDL

#	Step	Description
1	Configure the WebLogic Server domain for the required JMS components.	See Section 13.4, "Configuring the Host WebLogic Server Instance for the JMS Transport Web Service".
2	Update the WSDL to use JMS transport. (Optional)	This step is optional. If you do not update the WSDL to use JMS transport, then you must do at least one of the following: ■ Edit the stubbed out JWS file to add the `@WLJmsTransport` annotation to your JWS file, as described in Step 4. ■ Add a `<WLJmsTransport>` child element to the `jwsc` Ant task, as described in Step 5. See Section 13.7, "Updating the WSDL to Use JMS Transport".
3	Run the `wsdlc` Ant task against the WSDL file.	For example, if the target that calls the `wsdlc` Ant task is called generate-from-wsdl, then you would run: `prompt> ant generate-from-wsdl` See "Running the wsdlc WebLogic Web Services Ant Task" on page 3-11.
4	Update the stubbed-out JWS file.	The `wsdlc` Ant task generates a stubbed-out JWS file.You need to add your business code to the Web service so it behaves as you want. See "Updating the Stubbed-out JWS Implementation Class File Generated By wsdlc" on page 3-13 If you updated the WSDL to use the JMS transport in Step 2, the JWS file includes the `@WLJmsTransport` annotation that defines the JMS transport. If the `@WLJmsTransport` annotation is not included in the JWS file, you must do at least one of the following: ■ Edit the JWS file to add the `@WLJmsTransport` annotation to your JWS file, as described in Section 13.5, "Using the @WLJmsTransport JWS Annotation". ■ Add a `<WLJmsTransport>` child element to the `jwsc` Ant task, as described in Step 5.
5	Add a `<WLJmsTransport>` child element to the `jwsc` Ant task. (Optional)	Use the `<WLJmsTransport>` child element to override the transports defined in the JWS file. This step is required if the JWS file does not include the `@WLJmsTransport` annotation, as noted in Step 4. Otherwise, this step is optional. See Section 13.6, "Using the <WLJmsTransport> Child Element of the jwsc Ant Task" for details.
6	Run the `jwsc` Ant task against the JWS file to build the Web service.	Specify the artifacts generated by the `wsdlc` Ant task as well as your updated JWS implementation file, to generate an Enterprise Application that implements the Web service. See "Running the jwsc WebLogic Web Services Ant Task" on page 3-7.
7	Deploy the Web service to WebLogic Server.	See "Deploying and Undeploying WebLogic Web Services" on page 3-14.

See Section 13.8, "Invoking a WebLogic Web Service Using JMS Transport" for information about updating your client application to invoke the Web service using JMS transport.

13.4 Configuring the Host WebLogic Server Instance for the JMS Transport Web Service

Configuring the WebLogic Server instance on which the JMS transport Web service is deployed involves configuring JMS resources, such as JMS servers and modules, that are used internally by the Web services runtime.

You can configure these resources manually or you can use the Configuration Wizard to extend the WebLogic Server domain using a Web services-specific extension template. Using the Configuration Wizard greatly simplifies the required configuration steps; for details, see "Configuring Your Domain For Web Services Features" on page 3-2.

Notes: Alternatively, you can use WLST to configure the resources. For information about using WLST to extend the domain, see "Configuring Existing Domains" in *Understanding the WebLogic Scripting Tool*.

A domain that does not contain Web Services resources will still boot and operate correctly for non-Web services scenarios, and any Web Services scenario that does not involve asynchronous request and response. You will, however, see INFO messages in the server log indicating that asynchronous resources have not been configured and that the asynchronous response service for Web services has not been completely deployed.

If you prefer to configure the resources manually, perform the following steps.

Table 13–3 Steps to Configure Host WebLogic Server Instance Manually for the JMS Transport Web Service

#	Step	Description
1	Invoke the Administration Console for the domain that contains the host WebLogic Server instance.	To invoke the Administration Console in your browser, enter the following URL: `http://host:port/console` where ▪ *host* refers to the computer on which the Administration Server is running. ▪ *port* refers to the port number where the Administration Server is listening for connection requests. The default port number for the Administration server is 7001. See "Invoking the Administration Console" in *Understanding WebLogic Web Services for Oracle WebLogic Server*.

Table 13–3 (Cont.) Steps to Configure Host WebLogic Server Instance Manually for the JMS Transport Web Service

#	Step	Description
3	Create a JMS Server.	Create a JMS Server. If a JMS server already exists, you can use it if you do not want to create a new one. See "Create JMS servers" in *Oracle WebLogic Server Administration Console Online Help*.
4	Create JMS module and define queue.	Create a JMS module, and then define a JMS queue in the module. If a JMS module already exists, you can use it if you do not want to create a new one. Target the JMS queue to the JMS server you created in the preceding step. Be sure you specify that this JMS queue is local, typically by setting the local JNDI name. See "Create JMS system modules" and "Create queues in a system module" in *Oracle WebLogic Server Administration Console Online Help*. If you want the JMS transport Web service to use the default Web services queue, set the JNDI name of the JMS queue to `weblogic.wsee.DefaultQueue`. Otherwise, if you use a different JNDI name, be sure to specify the queue name when specifying the `@WLJmsTransport` annotation or `<WLJmsTransport>` child element of the jwsc Ant task. **Clustering Considerations:** If you are using the Web service JMS transport feature in a cluster, you must: ■ Create a *local* JMS queue, rather than a distributed queue, when creating the JMS queue. ■ Explicitly target this JMS queue to each server in the cluster.

13.5 Using the @WLJmsTransport JWS Annotation

If you know at the time that you program the JWS file that you want client applications to use JMS transport (instead of HTTP/S) to invoke the Web service, you can use the `@WLJmsTransport` to specify the details of the invocation. Later, at build-time, you can override the invocation defined in the JWS file and add additional JMS transport specifications, by specifying the `<WLJmsTransport>` child element of the `jwsc` Ant task, as described in Section 13.6, "Using the <WLJmsTransport> Child Element of the jwsc Ant Task".

Follow these guidelines when using the `@WLJmsTranport` annotation:

- You can include only *one* `@WLJmsTransport` annotation in a JWS file.
- Use the `queue` attribute to specify the JNDI name of the JMS queue you configured earlier in the section. If you want to use the default Web services queue (`weblogic.wsee.DefaultQueue`) then you do not have to specify the `queue` attribute.
- Use the `connectionFactory` attribute to specify the JNDI name of the connection factory. The default value of this attribute is the default JMS connection factory for your WebLogic Server instance.

The following example shows a simple JWS file that uses the `@WLJmsTransport` annotation, with the relevant code in **bold**:

```
package examples.webservices.jmstransport;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

import weblogic.jws.WLJmsTransport;
```

```
@WebService(name="JMSTransportPortType",
             serviceName="JMSTransportService",
            targetNamespace="http://example.org")

@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
             use=SOAPBinding.Use.LITERAL,
             parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)

// WebLogic-specific JWS annotation that specifies the context path and
// service URI used to build the URI of the Web Service is
// "transports/JMSTransport"

@WLJmsTransport(contextPath="transports", serviceUri="JMSTransport",
                queue="JMSTransportQueue", portName="JMSTransportServicePort",
             connectionFactory="JMSTransportConnectionFactory")

/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHello
 */

public class JMSTransportImpl {

  @WebMethod()
  public String sayHello(String message) {
    System.out.println("sayHello:" + message);
    return "Here is the message: '" + message + "'";
  }
}
```

13.6 Using the <WLJmsTransport> Child Element of the jwsc Ant Task

You can also specify the JMS transport at build-time by using the `<WLJmsTransport>` child element of the `<jws>` element of the `jwsc` Ant task. Reasons for specifying the transport at build-time include:

- You need to override the attribute values specified in the JWS file.
- The JWS file specifies a different transport, and at build-time you decide that JMS should be the transport.
- The JWS file does not include a `@WLXXXTransport` annotation; thus by default the HTTP transport is used, but at build-time you decide you want to clients to use the JMS transport to invoke the Web service.

If you specify a transport to the `jwsc` Ant task, it takes precedence over any transport annotation in the JWS file.

The following example shows how to specify a transport to the `jwsc` Ant task:

```
  <target name="build-service">

    <jwsc
      srcdir="src"
      destdir="${ear-dir}">
      <jws file="examples/webservices/jmstransport/JMSTransportImpl.java">

        <WLJmsTransport
                contextPath="transports"
                serviceUri="JMSTransport"
```

```
                portName="JMSTransportServicePort"
               queue="JMSTransportQueue"
                connectionFactory="JMSTransportConnectionFactory"/>

    </jws>

  </jwsc>

</target>
```

The preceding example shows how to specify the same values for the URL and JMS queue as were specified in the JWS file shown in Section 13.5, "Using the @WLJmsTransport JWS Annotation".

For more information about using the `jwsc` Ant task, see "jwsc" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

13.7 Updating the WSDL to Use JMS Transport

To update the WSDL to use JMS transport, you need to add `<wsdl:binding>` and `<wsdl:service>` definitions that define JMS transport information. You can add the definitions in one of the following ways:

- Edit the existing HTTP `<wsdl:binding>` and `<wsdl:service>` definitions.
- To specify multiple transport options in the WSDL, copy the existing HTTP `<wsdl:binding>` and `<wsdl:service>` definitions and edit them to use JMS transport.

In either case, you must modify the `<wsdl:binding>` and `<wsdl:service>` definitions to use JMS transport as follows:

- Set the `transport` attribute of the `<soapwsdl:binding>` child element of the `<wsdl:binding>` element to `http://www.openuri.org/2002/04/soap/jms/`. For example:

```
<binding name="JmsTransportServiceSoapBindingjms"
type="tns:JmsTransportPortType">
    <soap:binding style="document"
transport="http://www.openuri.org/2002/04/soap/jms/"/>
```

- Specify a JMS-style endpoint URL for the `location` attribute of the `<soapwsdl:address>` child element of the `<wsdl:service>`. For example:

```
<s0:service name="JmsTransportService">
    <s0:port binding="s1:JmsTransportServiceSoapBindingjms"
name="JmsTransportServicePort">
        <s2:address
location="jms://localhost:7001/transports/JmsTransport?URI=JMSTransportQueue"/>
    </s0:port>
  </s0:service>
```

13.8 Invoking a WebLogic Web Service Using JMS Transport

You write a client application to invoke a Web service using JMS transport in the same way as you write one using the HTTP transport; the only difference is that you must ensure that the JMS queue (specified by the `@WLJmsTransport` annotation or `<WLJmsTransport>` child element of the `jwsc` Ant task) and other JMS objects have been created. See Section 13.2, "Using JMS Transport Starting From Java: Main Steps"

or Section 13.3, "Using JMS Transport Starting From WSDL: Main Steps" for more information.

Although you cannot *invoke* a JMS-transport-configured Web service using HTTP, you can view its WSDL using HTTP, which is how the `clientgen` Ant task is still able to create the JAX-RPC stubs for the Web service. For example, the URL for the WSDL of the Web service shown in this section would be:

```
http://host:port/transports/JMSTransport?WSDL
```

However, because the endpoint address in the WSDL of the deployed Web service uses `jms://` instead of `http://`, and the address includes the qualifier `?URI=`*JMS_QUEUE*, the `clientgen` Ant task automatically creates the stubs needed to use the JMS transport when invoking the Web service, and your client application need not do anything different than normal. An example of a JMS endpoint address is as follows:

```
jms://host:port/transports/JMSTransport?URI=JMSTransportQueue
```

> **Note:** If you have specified that the Web service you invoke using JMS transport also runs within the context of a transaction (in other words, the JWS file includes the `@weblogic.jws.Transactional` annotation), you must use asynchronous request-response when invoking the service. If you do not, a deadlock will occur and the invocation will fail.

For general information about invoking a Web service, see "Developing JAX-RPC Web Service Clients" on page 6-1.

13.8.1 Overriding the Default Service Address URL

When you write a client application that uses the `clientgen`-generated JAX-RPC stubs to invoke a Web service, the default service address URL of the Web service is the one specified in the `<address>` element of the WSDL file argument of the `Service` constructor.

Sometimes, however, you might need to override this address, in particular when invoking a WebLogic Web service that is deployed to a cluster and you want to specify the cluster address or a list of addresses of the managed servers in the cluster. You might also want to use the `t3` protocol to invoke the Web service. To override this service endpoint URL when using JMS transport, use the `weblogic.wsee.jaxrpc.WLStub.JMS_TRANSPORT_JNDI_URL` stub property as shown in the following example:

```
package examples.webservices.jmstransport.client;

import weblogic.wsee.jaxrpc.WLStub;

import java.rmi.RemoteException;

import javax.xml.rpc.ServiceException;
import javax.xml.rpc.Stub;

/**
 * This is a simple standalone client application that invokes the
 * the <code>sayHello</code> operation of the JMSTransport Web service.
 */
```

```
public class Main {

  public static void main(String[] args)
      throws ServiceException, RemoteException{

    JMSTransportService service = new JMSTransportService_Impl(args[0] + "?WSDL"
);
    JMSTransportPortType port = service.getJMSTransportServicePort();

    Stub stub = (Stub) port;

    stub._setProperty(WLStub.JMS_TRANSPORT_JNDI_URL,
                      "t3://shackell01.amer.com:7001");
    try {
      String result = null;

      result = port.sayHello("Hi there! ");

      System.out.println( "Got JMS result: " + result );

    } catch (RemoteException e) {
      throw e;
    }
  }
}
```

See "WLStub" in *Java API Reference for Oracle WebLogic Server* for additional stub properties.

13.8.2 Using JMS BytesMessage Rather Than the Default TextMessage

When you use JMS transport, the Web services runtime uses, by default, the `javax.jms.TextMessage` object to send the message. This is usually adequate for most client applications, but sometimes you might need to send binary data rather than ordinary text; in this case you must request that the Web services runtime use `javax.jms.BytesMessage` instead. To do this, use the `WLStub.JMS_TRANSPORT_MESSAGE_TYPE` stub property in your client application and set it to the value `WLStub.JMS_BYTESMESSAGE`, as shown in the following example:

```
stub._setProperty(WLStub.JMS_TRANSPORT_MESSAGE_TYPE,
                  WLStub.JMS_BYTESMESSAGE);
```

The Web services runtime sends back the response using the same message data type as the request.

See Section 13.8.1, "Overriding the Default Service Address URL" for a full example of a client application in which you can set this property. See "WLStub" in *Java API Reference for Oracle WebLogic Server* for additional stub properties.

13.8.3 Disabling HTTP Access to the WSDL File

As described in Section 13.8, "Invoking a WebLogic Web Service Using JMS Transport", the WSDL of the deployed Web service is, by default, still accessible using HTTP. If you want to disable access to the WSDL file, in particular if your Web service can be accessed outside of a firewall, then you can do one of the following:

- Use the `weblogic.jws.WSDL` annotation in your JWS file to programmatically disable access. For details, see "weblogic.jws.WSDL" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

- Use the Administration Console to disable access to the WSDL file *after* the Web service has been deployed. In this case, the configuration information will be stored in the deployment plan rather than through the annotation.

  To use the Administration Console to perform this task, go to the Configuration -> General page of the deployed Web service and uncheck the **View Dynamic WSDL Enabled** check box. After saving the configuration to the deployment plan, you must redeploy (update) the Web service, or Enterprise Application which contains it, for the change to take effect.

14

Creating and Using SOAP Message Handlers

This chapter describes how to create and use SOAP message handlers with WebLogic Java API for XML-based RPC (JAX-RPC) Web services.

This chapter includes the following topics:

14.1 Overview of SOAP Message Handlers

Some Web services need access to the SOAP message, for which you can create SOAP message handlers.

A SOAP message handler provides a mechanism for intercepting the SOAP message in both the request and response of the Web service. You can create handlers in both the Web service itself and the client applications that invoke the Web service.

A simple example of using handlers is to access information in the header part of the SOAP message. You can use the SOAP header to store Web service specific information and then use handlers to manipulate it.

You can also use SOAP message handlers to improve the performance of your Web service. After your Web service has been deployed for a while, you might discover that many consumers invoke it with the same parameters. You could improve the performance of your Web service by caching the results of popular invokes of the Web service (assuming the results are static) and immediately returning these results when appropriate, without ever invoking the back-end components that implement the Web service. You implement this performance improvement by using handlers to check the request SOAP message to see if it contains the popular parameters.

The following table lists the standard JWS annotations that you can use in your JWS file to specify that a Web service has a handler chain configured; later sections discuss how to use the annotations in more detail. For additional information, see the Web services MetaData for the Java Platform (JSR-181) specification at `http://www.jcp.org/en/jsr/detail?id=181`.

Table 14–1 JWS Annotations Used To Configure SOAP Message Handler Chains

JWS Annotation	Description
`javax.jws.HandlerChain`	Associates the Web service with an externally defined handler chain. Use this annotation when multiple Web services need to share the same handler configuration, or if the handler chain consists of handlers for multiple transports.
`javax.jws.soap.SOAPMessageHandlers`	Specifies a list of SOAP handlers that run before and after the invocation of each Web service operation. Use this annotation (rather than `@HandlerChain`) if embedding handler configuration information in the JWS file itself is preferred, rather than having an external configuration file. The `@SOAPMessageHandler` annotation is an array of `@SOAPMessageHandlers`. The handlers are executed in the order they are listed in this array. **Note;** This annotation works with JAX-RPC Web services *only*.
`javax.jws.soap.SOAPMessageHandler`	Specifies a single SOAP message handler in the `@SOAPMessageHandlers` array.

The following table describes the main classes and interfaces of the `javax.xml.rpc.handler` API, some of which you use when creating the handler itself. These APIs are discussed in detail in a later section. For additional information about these APIs, see the JAX-RPC 1.1 specification at http://java.net/projects/jax-rpc/.

Table 14–2 JAX-RPC Handler Interfaces and Classes

javax.xml.rpc.handler Classes and Interfaces	Description
`Handler`	Main interface that is implemented when creating a handler. Contains methods to handle the SOAP request, response, and faults.
`GenericHandler`	Abstract class that implements the `Handler` interface. User should extend this class when creating a handler, rather than implement `Handler` directly. The `GenericHandler` class is a convenience abstract class that makes writing handlers easy. This class provides default implementations of the life cycle methods `init` and `destroy` and also different handle methods. A handler developer should only override methods that it needs to specialize as part of the derived handler implementation class.
`HandlerChain`	Interface that represents a list of handlers. An implementation class for the `HandlerChain` interface abstracts the policy and mechanism for the invocation of the registered handlers.
`HandlerRegistry`	Interface that provides support for the programmatic configuration of handlers in a HandlerRegistry.
`HandlerInfo`	Class that contains information about the handler in a handler chain. A `HandlerInfo` instance is passed in the `Handler.init` method to initialize a `Handler` instance.

Table 14–2 (Cont.) JAX-RPC Handler Interfaces and Classes

javax.xml.rpc.handler Classes and Interfaces	Description
`MessageContext`	Abstracts the message context processed by the handler. The `MessageContext` properties allow the handlers in a handler chain to share processing state.
`soap.SOAPMessageContext`	Sub-interface of the MessageContext interface used to get at or update the SOAP message.
`javax.xml.soap.SOAPMessage`	Object that contains the actual request or response SOAP message, including its header, body, and attachment.

14.2 Adding SOAP Message Handlers to a Web Service: Main Steps

The following procedure describes the high-level steps to add SOAP message handlers to your Web service.

It is assumed that you have created a basic JWS file that implements a Web service and that you want to update the Web service by adding SOAP message handlers and handler chains. It is also assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file that includes a target for running the `jwsc` Ant task. For more information, see the following sections:

- "Examples for JAX-RPC Web Service Developers" on page 2-1
- "Developing JAX-RPC Web Services" on page 3-1
- "Programming the JWS File" on page 4-1
- "Developing JAX-RPC Web Service Clients" on page 6-1

Table 14–3 Steps to Add SOAP Message Handlers to a Web Service

#	Step	Description
1	Design the handlers and handler chains.	See Section 14.3, "Designing the SOAP Message Handlers and Handler Chains".
2	For each handler in the handler chain, create a Java class that extends the `javax.xml.rpc.handler.GenericHandler` abstract class.	See Section 14.4, "Creating the GenericHandler Class".
3	Update your JWS file, adding annotations to configure the SOAP message handlers.	See Section 14.5, "Configuring Handlers in the JWS File".
4	If you are using the `@HandlerChain` standard annotation in your JWS file, create the handler chain configuration file.	See Section 14.6, "Creating the Handler Chain Configuration File".
5	Compile all handler classes in the handler chain and rebuild your Web service.	See Section 14.7, "Compiling and Rebuilding the Web Service".

For information about creating client-side SOAP message handlers and handler chains, see Section 14.8, "Creating and Using Client-Side SOAP Message Handlers".

14.3 Designing the SOAP Message Handlers and Handler Chains

When designing your SOAP message handlers and handler chains, you must decide:

- The number of handlers needed to perform all the work
- The sequence of execution

Each handler in a handler chain has one method for handling the request SOAP message and another method for handling the response SOAP message. An ordered group of handlers is referred to as a *handler chain*. You specify that a Web service has a handler chain attached to it with one of two JWS annotations: `@HandlerChain` or `@SOAPMessageHandler`. When to use which is discussed in a later section.

When invoking a Web service, WebLogic Server executes handlers as follows:

1. The `handleRequest()` methods of the handlers in the handler chain are all executed in the order specified by the JWS annotation. Any of these `handleRequest()` methods might change the SOAP message request.
2. When the `handleRequest()` method of the last handler in the handler chain executes, WebLogic Server invokes the back-end component that implements the Web service, passing it the final SOAP message request.
3. When the back-end component has finished executing, the `handleResponse()` methods of the handlers in the handler chain are executed in the *reverse* order specified in by the JWS annotation. Any of these `handleResponse()` methods might change the SOAP message response.
4. When the `handleResponse()` method of the first handler in the handler chain executes, WebLogic Server returns the final SOAP message response to the client application that invoked the Web service.

For example, assume that you are going to use the `@HandlerChain` JWS annotation in your JWS file to specify an external configuration file, and the configuration file defines a handler chain called `SimpleChain` that contains three handlers, as shown in the following sample:

```
<jwshc:handler-config xmlns:jwshc="http://www.bea.com/xml/ns/jws"
   xmlns:soap1="http://HandlerInfo.org/Server1"
   xmlns:soap2="http://HandlerInfo.org/Server2"
   xmlns="http://java.sun.com/xml/ns/j2ee" >

   <jwshc:handler-chain>

     <jwshc:handler-chain-name>SimpleChain</jwshc:handler-chain-name>

     <jwshc:handler>
       <handler-name>handlerOne</handler-name>
       <handler-class>examples.webservices.soap_handlers.global_
handler.ServerHandler1</handler-class>
     </jwshc:handler>

     <jwshc:handler>
       <handler-name>handlerTwo</handler-name>
       <handler-class>examples.webservices.soap_handlers.global_
handler.ServerHandler2</handler-class>
     </jwshc:handler>

     <jwshc:handler>
       <handler-name>handlerThree</handler-name>
       <handler-class>examples.webservices.soap_handlers.global_
handler.ServerHandler3</handler-class>
```

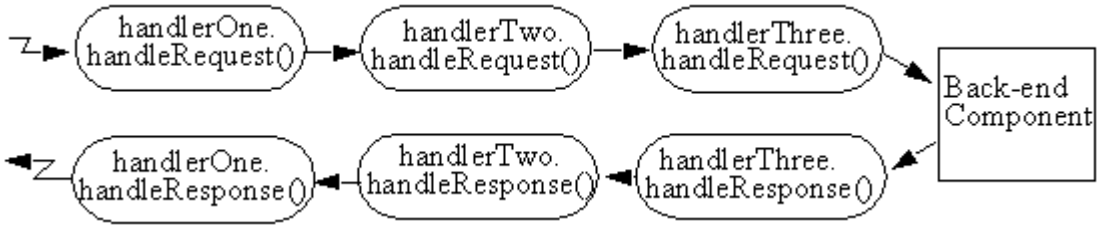
```
      </jwshc:handler>

    </jwshc:handler-chain>

</jwshc:handler-config>
```

The following graphic shows the order in which WebLogic Server executes the `handleRequest()` and `handleResponse()` methods of each handler.

Figure 14–1 Order of Execution of Handler Methods



Each SOAP message handler has a separate method to process the request and response SOAP message because the same type of processing typically must happen for the inbound and outbound message. For example, you might design an Encryption handler whose `handleRequest()` method decrypts secure data in the SOAP request and `handleResponse()` method encrypts the SOAP response.

You can, however, design a handler that process only the SOAP request and does no equivalent processing of the response.

You can also choose not to invoke the next handler in the handler chain and send an immediate response to the client application at any point.

14.4 Creating the GenericHandler Class

Your SOAP message handler class should extend the `javax.rpc.xml.handler.GenericHandler` abstract class, which itself implements the `javax.rpc.xml.handler.Handler` interface.

The `GenericHandler` class is a convenience abstract class that makes writing handlers easy. This class provides default implementations of the life cycle methods `init()` and `destroy()` and the various `handleXXX()` methods of the `Handler` interface. When you write your handler class, only override those methods that you need to customize as part of your `Handler` implementation class.

In particular, the `Handler` interface contains the following methods that you can implement in your handler class that extends `GenericHandler`:

- `init()`

  See Section 14.4.1, "Implementing the Handler.init() Method".

- `destroy()`

  See Section 14.4.2, "Implementing the Handler.destroy() Method".

- `getHeaders()`

  See Section 14.4.3, "Implementing the Handler.getHeaders() Method".

- `handleRequest()`

  See Section 14.4.4, "Implementing the Handler.handleRequest() Method".

- `handleResponse()`

  See Section 14.4.5, "Implementing the Handler.handleResponse() Method".

- `handleFault()`

  See Section 14.4.6, "Implementing the Handler.handleFault() Method".

Sometimes you might need to directly view or update the SOAP message from within your handler, in particular when handling attachments, such as image. In this case, use the `javax.xml.soap.SOAPMessage` abstract class, which is part of the SOAP With Attachments API for Java 1.1 (SAAJ) specification at https://saaj.dev.java.net/. For details, see Section 14.4.7, "Directly Manipulating the SOAP Request and Response Message Using SAAJ".

The following example demonstrates a simple SOAP message handler that prints out the SOAP request and response messages to the WebLogic Server log file:

```
package examples.webservices.soap_handlers.global_handler;

import javax.xml.namespace.QName;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.GenericHandler;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.rpc.JAXRPCException;

import weblogic.logging.NonCatalogLogger;

/**
 * This class implements a handler in the handler chain, used to access the SOAP
 * request and response message.
 * <p>
 * This class extends the <code>javax.xml.rpc.handler.GenericHandler</code>
 * abstract classs and simply prints the SOAP request and response messages to
 * the server log file before the messages are processed by the backend
 * Java class that implements the Web Service itself.
 */

public class ServerHandler1 extends GenericHandler {

  private NonCatalogLogger log;

  private HandlerInfo handlerInfo;

  /**
   *  Initializes the instance of the handler.  Creates a nonCatalogLogger to
   *  log messages to.
   */

  public void init(HandlerInfo hi) {

    log = new NonCatalogLogger("WebService-LogHandler");
    handlerInfo = hi;
  }

  /**
   * Specifies that the SOAP request message be logged to a log file before the
   * message is sent to the Java class that implements the Web Service.
   */

  public boolean handleRequest(MessageContext context) {
```

```
    SOAPMessageContext messageContext = (SOAPMessageContext) context;

    System.out.println("** Request: "+messageContext.getMessage().toString());
    log.info(messageContext.getMessage().toString());
    return true;

  }

  /**
   * Specifies that the SOAP response message be logged to a log file before the
   * message is sent back to the client application that invoked the Web
   * service.
   */

  public boolean handleResponse(MessageContext context) {

    SOAPMessageContext messageContext = (SOAPMessageContext) context;

    System.out.println("** Response: "+messageContext.getMessage().toString());
    log.info(messageContext.getMessage().toString());
    return true;

  }

  /**
   * Specifies that a message be logged to the log file if a SOAP fault is
   * thrown by the Handler instance.
   */

  public boolean handleFault(MessageContext context) {

    SOAPMessageContext messageContext = (SOAPMessageContext) context;

    System.out.println("** Fault: "+messageContext.getMessage().toString());
    log.info(messageContext.getMessage().toString());
        return true;

  }

  public QName[] getHeaders() {

    return handlerInfo.getHeaders();

  }

}
```

14.4.1 Implementing the Handler.init() Method

The `Handler.init()` method is called to create an instance of a `Handler` object and to enable the instance to initialize itself. Its signature is:

```
public void init(HandlerInfo config) throws JAXRPCException {}
```

The `HandlerInfo` object contains information about the SOAP message handler, in particular the initialization parameters. Use the `HandlerInfo.getHandlerConfig()` method to get the parameters; the method returns a `java.util.Map` object that contains name-value pairs.

Implement the `init()` method if you need to process the initialization parameters or if you have other initialization tasks to perform.

Sample uses of initialization parameters are to turn debugging on or off, specify the name of a log file to which to write messages or errors, and so on.

14.4.2 Implementing the Handler.destroy() Method

The `Handler.destroy()` method is called to destroy an instance of a `Handler` object. Its signature is:

```
public void destroy() throws JAXRPCException {}
```

Implement the `destroy()` method to release any resources acquired throughout the handler's life cycle.

14.4.3 Implementing the Handler.getHeaders() Method

The `Handler.getHeaders()` method gets the header blocks that can be processed by this `Handler` instance. Its signature is:

```
public QName[] getHeaders() {}
```

14.4.4 Implementing the Handler.handleRequest() Method

The `Handler.handleRequest()` method is called to intercept a SOAP message request before it is processed by the back-end component. Its signature is:

```
public boolean handleRequest(MessageContext mc)
        throws JAXRPCException,SOAPFaultException  {}
```

Implement this method to perform such tasks as decrypting data in the SOAP message before it is processed by the back-end component, and so on.

The `MessageContext` object abstracts the message context processed by the SOAP message handler. The `MessageContext` properties allow the handlers in a handler chain to share processing state.

Use the `SOAPMessageContext` sub-interface of `MessageContext` to get at or update the contents of the SOAP message request. The SOAP message request itself is stored in a `javax.xml.soap.SOAPMessage` object. For detailed information on this object, see Section 14.4.7, "Directly Manipulating the SOAP Request and Response Message Using SAAJ".

The `SOAPMessageContext` class defines two methods for processing the SOAP request:

- `SOAPMessageContext.getMessage()` returns a `javax.xml.soap.SOAPMessage` object that contains the SOAP message request.
- `SOAPMessageContext.setMessage(javax.xml.soap.SOAPMessage)` updates the SOAP message request after you have made changes to it.

After you code all the processing of the SOAP request, code one of the following scenarios:

- Invoke the next handler on the handler request chain by returning `true`.

  The next handler on the request chain is specified as either the next `<handler>` subelement of the `<handler-chain>` element in the configuration file specified by the `@HandlerChain` annotation, or the next `@SOAPMessageHandler` in the

array specified by the `@SOAPMessageHandlers` annotation. If there are no more handlers in the chain, the method either invokes the back-end component, passing it the final SOAP message request, or invokes the `handleResponse()` method of the last handler, depending on how you have configured your Web service.

- Block processing of the handler request chain by returning `false`.

  Blocking the handler request chain processing implies that the back-end component does not get executed for this invoke of the Web service. You might want to do this if you have cached the results of certain invokes of the Web service, and the current invoke is on the list.

  Although the handler request chain does not continue processing, WebLogic Server does invoke the handler *response* chain, starting at the current handler. For example, assume that a handler chain consists of two handlers: handlerA and handlerB, where the `handleRequest()` method of handlerA is invoked before that of handlerB. If processing is blocked in handlerA (and thus the `handleRequest()` method of handlerB is *not* invoked), the handler response chain starts at handlerA and the `handleRequest()` method of handlerB is not invoked either.

- Throw the `javax.xml.rpc.soap.SOAPFaultException` to indicate a SOAP fault.

  If the `handleRequest()` method throws a `SOAPFaultException`, WebLogic Server catches the exception, terminates further processing of the handler request chain, and invokes the `handleFault()` method of this handler.

- Throw a `JAXRPCException` for any handler-specific runtime errors.

  If the `handleRequest()` method throws a `JAXRPCException`, WebLogic Server catches the exception, terminates further processing of the handler request chain, logs the exception to the WebLogic Server log file, and invokes the `handleFault()` method of this handler.

14.4.5 Implementing the Handler.handleResponse() Method

The `Handler.handleResponse()` method is called to intercept a SOAP message response after it has been processed by the back-end component, but before it is sent back to the client application that invoked the Web service. Its signature is:

```
public boolean handleResponse(MessageContext mc) throws JAXRPCException {}
```

Implement this method to perform such tasks as encrypting data in the SOAP message before it is sent back to the client application, to further process returned values, and so on.

The `MessageContext` object abstracts the message context processed by the SOAP message handler. The `MessageContext` properties allow the handlers in a handler chain to share processing state.

Use the `SOAPMessageContext` sub-interface of `MessageContext` to get at or update the contents of the SOAP message response. The SOAP message response itself is stored in a `javax.xml.soap.SOAPMessage` object. See Section 14.4.7, "Directly Manipulating the SOAP Request and Response Message Using SAAJ".

The `SOAPMessageContext` class defines two methods for processing the SOAP response:

- `SOAPMessageContext.getMessage()`: returns a `javax.xml.soap.SOAPMessage` object that contains the SOAP message response.
- `SOAPMessageContext.setMessage(javax.xml.soap.SOAPMessage)`: updates the SOAP message response after you have made changes to it.

After you code all the processing of the SOAP response, code one of the following scenarios:

- Invoke the next handler on the handler response chain by returning `true`.

  The next response on the handler chain is specified as either the preceding `<handler>` subelement of the `<handler-chain>` element in the configuration file specified by the `@HandlerChain` annotation, or the preceding `@SOAPMessageHandler` in the array specified by the `@SOAPMessageHandlers` annotation. (Remember that responses on the handler chain execute in the *reverse* order that they are specified in the JWS file. See Section 14.3, "Designing the SOAP Message Handlers and Handler Chains" for more information.)

  If there are no more handlers in the chain, the method sends the final SOAP message response to the client application that invoked the Web service.

- Block processing of the handler response chain by returning `false`.

  Blocking the handler response chain processing implies that the remaining handlers on the response chain do not get executed for this invoke of the Web service and the current SOAP message is sent back to the client application.

- Throw a `JAXRPCException` for any handler specific runtime errors.

  If the `handleRequest()` method throws a `JAXRPCException`, WebLogic Server catches the exception, terminates further processing of the handler request chain, logs the exception to the WebLogic Server logfile, and invokes the `handleFault()` method of this handler.

14.4.6 Implementing the Handler.handleFault() Method

The `Handler.handleFault()` method processes the SOAP faults based on the SOAP message processing model. Its signature is:

```
public boolean handleFault(MessageContext mc) throws JAXRPCException  {}
```

Implement this method to handle processing of any SOAP faults generated by the `handleResponse()` and `handleRequest()` methods, as well as faults generated by the back-end component.

The `MessageContext` object abstracts the message context processed by the SOAP message handler. The `MessageContext` properties allow the handlers in a handler chain to share processing state.

Use the `SOAPMessageContext` sub-interface of `MessageContext` to get at or update the contents of the SOAP message. The SOAP message itself is stored in a `javax.xml.soap.SOAPMessage` object. See Section 14.4.7, "Directly Manipulating the SOAP Request and Response Message Using SAAJ".

The `SOAPMessageContext` class defines the following two methods for processing the SOAP message:

- `SOAPMessageContext.getMessage()`: returns a `javax.xml.soap.SOAPMessage` object that contains the SOAP message.

- `SOAPMessageContext.setMessage(javax.xml.soap.SOAPMessage)`: updates the SOAP message after you have made changes to it.

After you code all the processing of the SOAP fault, do one of the following:

- Invoke the `handleFault()` method on the next handler in the handler chain by returning `true`.
- Block processing of the handler fault chain by returning `false`.

14.4.7 Directly Manipulating the SOAP Request and Response Message Using SAAJ

The `javax.xml.soap.SOAPMessage` abstract class is part of the SOAP With Attachments API for Java 1.1 (SAAJ) specification (see https://saaj.dev.java.net/). You use the class to manipulate request and response SOAP messages when creating SOAP message handlers. This section describes the basic structure of a `SOAPMessage` object and some of the methods you can use to view and update a SOAP message.

A `SOAPMessage` object consists of a `SOAPPart` object (which contains the actual SOAP XML document) and zero or more attachments.

Refer to the SAAJ Javadocs for the full description of the `SOAPMessage` class.

14.4.7.1 The SOAPPart Object

The `SOAPPart` object contains the XML SOAP document inside of a `SOAPEnvelope` object. You use this object to get the actual SOAP headers and body.

The following sample Java code shows how to retrieve the SOAP message from a `MessageContext` object, provided by the `Handler` class, and get at its parts:

```
SOAPMessage soapMessage =  messageContext.getMessage();
SOAPPart soapPart = soapMessage.getSOAPPart();
SOAPEnvelope soapEnvelope = soapPart.getEnvelope();
SOAPBody soapBody = soapEnvelope.getBody();
SOAPHeader soapHeader = soapEnvelope.getHeader();
```

14.4.7.2 The AttachmentPart Object

The `javax.xml.soap.AttachmentPart` object contains the optional attachments to the SOAP message. Unlike the rest of a SOAP message, an attachment is not required to be in XML format and can therefore be anything from simple text to an image file.

Note: If you are going to access a `java.awt.Image` attachment from your SOAP message handler, see Section 14.4.7.3, "Manipulating Image Attachments in a SOAP Message Handler" for important information.

Use the following methods of the `SOAPMessage` class to manipulate the attachments:

- `countAttachments()`: returns the number of attachments in this SOAP message.
- `getAttachments()`: retrieves all the attachments (as `AttachmentPart` objects) into an `Iterator` object.
- `createAttachmentPart()`: create an `AttachmentPart` object from another type of `Object`.
- `addAttachmentPart()`: adds an `AttachmentPart` object, after it has been created, to the `SOAPMessage`.

14.4.7.3 Manipulating Image Attachments in a SOAP Message Handler

It is assumed in this section that you are creating a SOAP message handler that accesses a `java.awt.Image` attachment and that the `Image` has been sent from a client application that uses the client JAX-RPC stubs generated by the `clientgen` Ant task.

In the client code generated by the `clientgen` Ant task, a `java.awt.Image` attachment is sent to the invoked WebLogic Web service with a MIME type of `text/xml` rather than `image/gif`, and the image is serialized into a stream of integers that represents the image. In particular, the client code serializes the image using the following format:

- `int width`
- `int height`
- `int[] pixels`

This means that, in your SOAP message handler that manipulates the received Image attachment, you must deserialize this stream of data to then re-create the original image.

14.5 Configuring Handlers in the JWS File

There are two standard annotations you can use in your JWS file to configure a handler chain for a Web service: `@javax.jws.HandlerChain` and `@javax.jws.soap.SOAPMessageHandlers`.

14.5.1 @javax.jws.HandlerChain

When you use the `@javax.jws.HandlerChain` annotation (also called `@HandlerChain` in this chapter for simplicity) you use the `file` attribute to specify an external file that contains the configuration of the handler chain you want to associate with the Web service. The configuration includes the list of handlers in the chain, the order in which they execute, the initialization parameters, and so on.

Use the `@HandlerChain` annotation, rather than the `@SOAPMessageHandlers` annotation, in your JWS file if one or more of the following conditions apply:

- You want multiple Web services to share the same configuration.
- Your handler chain includes handlers for multiple transports.
- You want to be able to change the handler chain configuration for a Web service without recompiling the JWS file that implements it.

The following JWS file shows an example of using the `@HandlerChain` annotation; the relevant Java code is shown in bold:

```
package examples.webservices.soap_handlers.global_handler;

import java.io.Serializable;

import javax.jws.HandlerChain;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;

import weblogic.jws.WLHttpTransport;

@WebService(serviceName="HandlerChainService",
```

```
            name="HandlerChainPortType")

// Standard JWS annotation that specifies that the handler chain called
// "SimpleChain", configured in the HandlerConfig.xml file, should fire
// each time an operation of the Web Service is invoked.

@HandlerChain(file="HandlerConfig.xml", name="SimpleChain")

@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
             use=SOAPBinding.Use.LITERAL,
             parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)

@WLHttpTransport(contextPath="HandlerChain", serviceUri="HandlerChain",
                 portName="HandlerChainServicePort")

/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHello.  The Web Service also
 * has a handler chain associated with it, as specified by the
 * @HandlerChain annotation.
 */

public class HandlerChainImpl  {

   public String sayHello(String input) {
     weblogic.utils.Debug.say( "in backend component. input:" +input );
     return "'" + input + "' to you too!";
   }
}
```

Before you use the `@HandlerChain` annotation, you must import it into your JWS file, as shown in the preceding example.

Use the `file` attribute of the `@HandlerChain` annotation to specify the name of the external file that contains configuration information for the handler chain. The value of this attribute is a URL, which may be relative or absolute. Relative URLs are relative to the location of the JWS file at the time you run the `jwsc` Ant task to compile the file.

Use the `name` attribute to specify the name of the handler chain in the configuration file that you want to associate with the Web service. The value of this attribute corresponds to the `name` attribute of the `<handler-chain>` element in the configuration file.

> **Note:** It is an error to specify more than one `@HandlerChain` annotation in a single JWS file. It is also an error to combine the `@HandlerChain` annotation with the `@SOAPMessageHandlers` annotation.

For details about creating the external configuration file, see Section 14.6, "Creating the Handler Chain Configuration File".

For additional detailed information about the standard JWS annotations discussed in this section, see the Web services Metadata for the Java Platform specification at `http://www.jcp.org/en/jsr/detail?id=181`.

14.5.2 @javax.jws.soap.SOAPMessageHandlers

> **Note:** This annotation has been deprecated as of the Web services Metadata for the Java Platform specification (JSR-181) at http://www.jcp.org/en/jsr/detail?id=181.

When you use the `@javax.jws.soap.SOAPMessageHandlers` (also called `@SOAPMessageHandlers` in this section for simplicity) annotation, you specify, within the JWS file itself, an array of SOAP message handlers (specified with the `@SOAPMessageHandler` annotation) that execute before and after the operations of a Web service. The `@SOAPMessageHandler` annotation includes attributes to specify the class name of the handler, the initialization parameters, list of SOAP headers processed by the handler, and so on. Because you specify the list of handlers within the JWS file itself, the configuration of the handler chain is embedded within the Web service.

Use the `@SOAPMessageHandlers` annotation if one or more of the following conditions apply:

- You prefer to embed the configuration of the handler chain inside the Web service itself, rather than specify the configuration in an external file.
- Your handler chain includes only SOAP handlers and none for any other transport.
- You prefer to recompile the JWS file each time you change the handler chain configuration.

The following JWS file shows a simple example of using the `@SOAPMessageHandlers` annotation; the relevant Java code is shown in bold:

```
package examples.webservices.soap_handlers.simple;

import java.io.Serializable;

import javax.jws.soap.SOAPMessageHandlers;
import javax.jws.soap.SOAPMessageHandler;
import javax.jws.soap.SOAPBinding;
import javax.jws.WebService;
import javax.jws.WebMethod;

import weblogic.jws.WLHttpTransport;

@WebService(name="SimpleChainPortType",
            serviceName="SimpleChainService")

// Standard JWS annotation that specifies a list of SOAP message handlers
// that exeucte before and after an invocation of all operations in the
// Web Serice.

@SOAPMessageHandlers ( {
        @SOAPMessageHandler (
          className="examples.webservices.soap_handlers.simple.ServerHandler1"),
        @SOAPMessageHandler (
          className="examples.webservices.soap_handlers.simple.ServerHandler2")
        } )

@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
             use=SOAPBinding.Use.LITERAL,
```

```
            parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)

@WLHttpTransport(contextPath="SimpleChain", serviceUri="SimpleChain",
                 portName="SimpleChainServicePort")

/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHello.  The Web Service also
 * has a handler chain associated with it, as specified by the
 * @SOAPMessageHandler/s annotations.
 */

public class SimpleChainImpl  {

   // by default all public methods are exposed as operations

   public String sayHello(String input)  {
     weblogic.utils.Debug.say( "in backend component. input:" +input );
     return "'" + input + "' to you too!";
   }
}
```

Before you use the `@SOAPMessageHandlers` and `@SOAPMessageHandler` annotations, you must import them into your JWS file, as shown in the preceding example. Note that these annotations are in the `javax.jws.soap` package.

The order in which you list the handlers (using the `@SOAPMessageHandler` annotation) in the `@SOAPMessageHandlers` array specifies the order in which the handlers execute: in forward order before the operation, and in reverse order after the operation. The preceding example configures two handlers in the handler chain, whose class names are `examples.webservices.soap_handlers.simple.ServerHandler1` and `examples.webservices.soap_handlers.simple.ServerHandler2`.

Use the `initParams` attribute of `@SOAPMessageHandler` to specify an array of initialization parameters expected by a particular handler. Use the `@InitParam` standard JWS annotation to specify the name/value pairs, as shown in the following example:

```
@SOAPMessageHandler(
  className = "examples.webservices.soap_handlers.simple.ServerHandler1",
  initParams = { @InitParam(name="logCategory", value="MyService")}
)
```

The `@SOAPMessageHandler` annotation also includes the `roles` attribute for listing the SOAP roles implemented by the handler, and the `headers` attribute for listing the SOAP headers processed by the handler.

> **Note:** It is an error to combine the `@SOAPMessageHandlers` annotation with the `@HandlerChain` annotation.

For additional detailed information about the standard JWS annotations discussed in this section, see the Web services Metadata for the Java Platform specification http://www.jcp.org/en/jsr/detail?id=181.

14.6 Creating the Handler Chain Configuration File

If you decide to use the `@HandlerChain` annotation in your JWS file to associate a handler chain with a Web service, you must create an external configuration file that specifies the list of handlers in the handler chain, the order in which they execute, the initialization parameters, and so on.

Because this file is external to the JWS file, you can configure multiple Web services to use this single configuration file to standardize the handler configuration file for all Web services in your enterprise. Additionally, you can change the configuration of the handler chains without needing to recompile all your Web services. Finally, if you include handlers in your handler chain that use a non-SOAP transport, then you are required to use the `@HandlerChain` annotation rather than the `@SOAPMessageHandler` annotation.

The configuration file uses XML to list one or more handler chains, as shown in the following simple example:

```
<jwshc:handler-config xmlns:jwshc="http://www.bea.com/xml/ns/jws"
   xmlns:soap1="http://HandlerInfo.org/Server1"
   xmlns:soap2="http://HandlerInfo.org/Server2"
   xmlns="http://java.sun.com/xml/ns/j2ee" >
   <jwshc:handler-chain>
     <jwshc:handler-chain-name>SimpleChain</jwshc:handler-chain-name>
     <jwshc:handler>
       <handler-name>handler1</handler-name>
       <handler-class>examples.webservices.soap_handlers.global_
handler.ServerHandler1</handler-class>
     </jwshc:handler>
     <jwshc:handler>
       <handler-name>handler2</handler-name>
       <handler-class>examples.webservices.soap_handlers.global_
handler.ServerHandler2</handler-class>
     </jwshc:handler>
   </jwshc:handler-chain>
</jwshc:handler-config>
```

In the example, the handler chain called `SimpleChain` contains two handlers: `handler1` and `handler2`, implemented with the class names specified with the `<handler-class>` element. The two handlers execute in forward order before the relevant Web service operation executes, and in reverse order after the operation executes.

Use the `<init-param>`, `<soap-role>`, and `<soap-header>` child elements of the `<handler>` element to specify the handler initialization parameters, SOAP roles implemented by the handler, and SOAP headers processed by the handler, respectively.

For the XML Schema that defines the external configuration file, additional information about creating it, and additional examples, see the Web services Metadata for the Java Platform specification at `http://www.jcp.org/en/jsr/detail?id=181`.

14.7 Compiling and Rebuilding the Web Service

It is assumed in this section that you have a working `build.xml` Ant file that compiles and builds your Web service, and you want to update the build file to include handler chain. See "Developing JAX-RPC Web Services" on page 3-1 for information on creating this `build.xml` file.

Follow these guidelines to update your development environment to include message handler compilation and building:

- After you have updated the JWS file with either the `@HandlerChain` or `@SOAPMessageHandlers` annotation, you must rerun the `jwsc` Ant task to recompile the JWS file and generate a new Web service. This is true anytime you make a change to an annotation in the JWS file.

  If you used the `@HandlerChain` annotation in your JWS file, reran the `jwsc` Ant task to regenerate the Web service, and subsequently changed only the external configuration file, you do not need to rerun `jwsc` for the second change to take affect.

- The `jwsc` Ant task compiles SOAP message handler Java files into handler classes (and then packages them into the generated application) if all the following conditions are true:

  – The handler classes are referenced in the `@HandlerChain` or `@SOAPMessageHandler(s)` annotations of the JWS file.

  – The Java files are located in the directory specified by the `sourcepath` attribute.

  – The classes are not currently in your CLASSPATH.

  If you want to compile the handler classes yourself, rather than let `jwsc` compile them automatically, ensure that the compiled classes are in your CLASSPATH before you run the `jwsc` Ant task.

- You deploy and invoke a Web service that has a handler chain associated with it in the same way you deploy and invoke one that has no handler chain. The only difference is that when you invoke any operation of the Web service, the WebLogic Web services runtime executes the handlers in the handler chain both before and after the operation invoke.

14.8 Creating and Using Client-Side SOAP Message Handlers

The preceding sections describe how to create server-side SOAP message handlers that execute as part of the Web service running on WebLogic Server. You can also create client-side handlers that execute as part of the client application that *invokes* a Web service operation. In the case of a client-side handler, the handler executes twice:

- Directly before the client application sends the SOAP request to the Web service
- Directly after the client application receives the SOAP response from the Web service

You can configure client-side SOAP message handlers for both stand-alone clients and clients that run inside of WebLogic Server.

You create the actual Java client-side handler in the same way you create a server-side handler: write a Java class that extends the `javax.xml.rpc.handler.GenericHandler` abstract class. In many cases you can use the exact same handler class on both the Web service running on WebLogic Server *and* the client applications that invoke the Web service. For example, you can write a generic logging handler class that logs all sent and received SOAP messages, both for the server and for the client.

Similar to the server-side SOAP handler programming, you use an XML file to specify to the `clientgen` Ant task that you want to invoke client-side SOAP message

handlers. However, the XML Schema of this XML file is slightly different, as described in the following procedure.

14.8.1 Using Client-Side SOAP Message Handlers: Main Steps

The following procedure describes the high-level steps to add client-side SOAP message handlers to the client application that invokes a Web service operation.

It is assumed that you have created the client application that invokes a deployed Web service, and that you want to update the client application by adding client-side SOAP message handlers and handler chains. It is also assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file that includes a target for running the `clientgen` Ant task. For more information, see "Developing JAX-RPC Web Service Clients" on page 6-1.

1. Design the client-side SOAP handlers and the handler chain which specifies the order in which they execute. This step is almost exactly the same as that of designing the server-side SOAP message handlers, except the perspective is from the client application, rather than a Web service.

   See Section 14.3, "Designing the SOAP Message Handlers and Handler Chains".

2. For each handler in the handler chain, create a Java class that extends the `javax.xml.rpc.handler.GenericHandler` abstract class. This step is very similar to the corresponding server-side step, except that the handler executes in a chain in the client rather than the server.

   See Section 14.4, "Creating the GenericHandler Class" for details about programming a handler class. See Section 14.8.2, "Example of a Client-Side Handler Class" for an example.

3. Create the client-side SOAP handler configuration file. This XML file describes the handlers in the handler chain, the order in which they execute, and any initialization parameters that should be sent.

   See Section 14.8.3, "Creating the Client-Side SOAP Handler Configuration File".

4. Update the `build.xml` file that builds your client application, specifying to the `clientgen` Ant task the name of the SOAP handler configuration file. Also ensure that the `build.xml` file compiles the handler files into Java classes and makes them available to your client application.

   See Section 14.8.5, "Specifying the Client-Side SOAP Handler Configuration File to clientgen".

5. Rebuild your client application by running the relevant task:

   ```
   prompt> ant build-client
   ```

When you next run the client application, the SOAP messaging handlers listed in the configuration file automatically execute before the SOAP request message is sent and after the response is received.

Note: You do *not* have to update your actual client application to invoke the client-side SOAP message handlers; as long as you specify to the `clientgen` Ant task the handler configuration file, the generated JAX-RPC stubs automatically take care of executing the handlers in the correct sequence.

14.8.2 Example of a Client-Side Handler Class

The following example shows a simple SOAP message handler class that you can configure for a client application that invokes a Web service.

```
package examples.webservices.client_handler.client;

import javax.xml.namespace.QName;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.GenericHandler;
import javax.xml.rpc.handler.MessageContext;

public class ClientHandler1 extends GenericHandler {

  private QName[] headers;

  public void init(HandlerInfo hi) {
    System.out.println("in " + this.getClass() + " init()");
  }

  public boolean handleRequest(MessageContext context) {
    System.out.println("in " + this.getClass() + " handleRequest()");
    return true;
  }

  public boolean handleResponse(MessageContext context) {
    System.out.println("in " + this.getClass() + " handleResponse()");
    return true;
  }

  public boolean handleFault(MessageContext context) {
    System.out.println("in " + this.getClass() + " handleFault()");
    return true;
  }

  public QName[] getHeaders() {
    return headers;
  }
}
```

14.8.3 Creating the Client-Side SOAP Handler Configuration File

The client-side SOAP handler configuration file specifies the list of handlers in the handler chain, the order in which they execute, the initialization parameters, and so on. See Section 14.8.4, "XML Schema for the Client-Side Handler Configuration File" for a full description of this file.

The configuration file uses XML to describe a single handler chain that contains one or more handlers, as shown in the following simple example:

```
<weblogic-wsee-clientHandlerChain
  xmlns="http://www.bea.com/ns/weblogic/90"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:j2ee="http://java.sun.com/xml/ns/j2ee">

        <handler>
          <j2ee:handler-name>clienthandler1</j2ee:handler-name>
          <j2ee:handler-class>examples.webservices.client_
handler.client.ClientHandler1</j2ee:handler-class>
          <j2ee:init-param>
            <j2ee:param-name>ClientParam1</j2ee:param-name>
```

```
            <j2ee:param-value>value1</j2ee:param-value>
          </j2ee:init-param>
        </handler>

        <handler>
          <j2ee:handler-name>clienthandler2</j2ee:handler-name>
          <j2ee:handler-class>examples.webservices.client_
handler.client.ClientHandler2</j2ee:handler-class>
        </handler>

</weblogic-wsee-clientHandlerChain>
```

In the example, the handler chain contains two handlers: `clienthandler1` and `clienthandler2`, implemented with the class names specified with the `<j2ee:handler-class>` element. The two handlers execute in forward order directly before the client application sends the SOAP request to the Web service, and then in reverse order directly after the client application receives the SOAP response from the Web service.

The example also shows how to use the `<j2ee:init-param>` element to specify one or more initialization parameters to a handler.

Use the `<soap-role>`, `<soap-header>`, and `<port-name>` child elements of the `<handler>` element to specify the SOAP roles implemented by the handler, the SOAP headers processed by the handler, and the port-name element in the WSDL with which the handler is associated with, respectively.

14.8.4 XML Schema for the Client-Side Handler Configuration File

The following XML Schema file defines the structure of the client-side SOAP handler configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>

<schema
  targetNamespace="http://www.bea.com/ns/weblogic/90"
  xmlns:wls="http://www.bea.com/ns/weblogic/90"
  xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  >
  <include schemaLocation="weblogic-j2ee.xsd"/>

  <element name="weblogic-wsee-clientHandlerChain"
           type="wls:weblogic-wsee-clientHandlerChainType">
       <xsd:key name="wsee-clienthandler-name-key">
            <xsd:annotation>
            <xsd:documentation>

            Defines the name of the handler. The name must be unique within the
            chain.

            </xsd:documentation>
            </xsd:annotation>
                <xsd:selector xpath="j2ee:handler"/>
                <xsd:field xpath="j2ee:handler-name"/>
       </xsd:key>
  </element>
```

```
  <complexType name="weblogic-wsee-clientHandlerChainType">
      <sequence>
        <xsd:element name="handler"
                     type="j2ee:service-ref_handlerType"
                     minOccurs="0" maxOccurs="unbounded">
        </xsd:element>
      </sequence>
  </complexType>
</schema>
```

A single configuration file specifies a single client-side handler chain. The root of the configuration file is `<weblogic-wsee-clientHandlerChain>`, and the file contains zero or more `<handler>` child elements, each of which describes a handler in the chain.

The structure of the `<handler>` element is described by the Java EE service-`ref_handlerType` complex type, specified in the Java EE 1.4 Web service client XML Schema http://java.sun.com/xml/ns/j2ee/j2ee_web_services_client_1_1.xsd.

14.8.5 Specifying the Client-Side SOAP Handler Configuration File to clientgen

Use the `handlerChainFile` attribute of the `clientgen` Ant task to specify the client-side SOAP handler configuration file, as shown in the following excerpt from a `build.xml` file:

```
<clientgen
  wsdl="http://ariel:7001/handlers/ClientHandlerService?WSDL"
  destDir="${clientclass-dir}"
  handlerChainFile="ClientHandlerChain.xml"
  packageName="examples.webservices.client_handler.client"/>
```

The JAX-RPC stubs generated by `clientgen` automatically ensure that the handlers described by the configuration file execute in the correct order before and after the client application invokes the Web service operation.

15 Using Database Web Services

This chapter describes how to use database Web services.

This chapter includes the following topics:

- Section 15.1, "Overview of Database Web Services"
- Section 15.2, "Type Mapping Between SQL and XML"

15.1 Overview of Database Web Services

> **Note:** With the removal of JAX-RPC support in Oracle JDeveloper, there is no longer development time support for PL/SQL database Web services; however, they will continue to be supported in the WebLogic Server rutime environment.
>
> You can create Oracle TopLink database JAX-WS Web service providers at design time, as described in "Creating TopLink Database Web Service Providers" in *Developing Applications with Oracle JDeveloper*.

In heterogeneous and disconnected environments, there is an increasing need to access stored procedures, data and metadata, through Web service interfaces. Database Web service technology enables Web services for databases. It works in two directions:

- Database Call-in—Access database resources as a Web service
- Database Call-out—Consuming external Web services from the database itself

15.1.1 Database Call-in

Turning the Oracle database into a Web service provider takes advantage of your investment in Java stored procedures, PL/SQL packages, Advanced Queues, pre-defined SQL queries and DML.

> **Note:** Creating Web services out of Query, Java, DML, and Advanced Queues is not supported in this release.

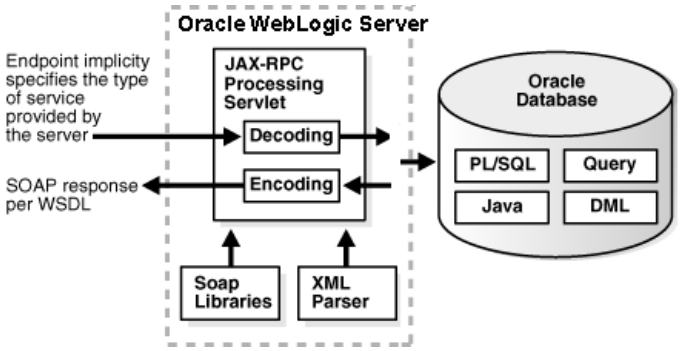
Client applications can query and retrieve data from Oracle databases and invoke stored procedures using standard Web service protocols. There is no dependency on Oracle specific database connectivity protocols. Applications can employ any cached

WebLogic Server connection. This approach is very beneficial in heterogeneous, distributed, and non-connected environments.

Since database Web services are a part of WebLogic Web services, they can participate in a consistent and uniform development and deployment environment. Messages exchanged between the Web service exposed database and the Web service client can take advantage of all of the management features provided by WebLogic Web services, such as security, reliability, auditing and logging.

The following figure illustrates Web service call-in.

Figure 15–1 Web Service Calling in to the Database



The following steps describe the process shown in the previous figure:

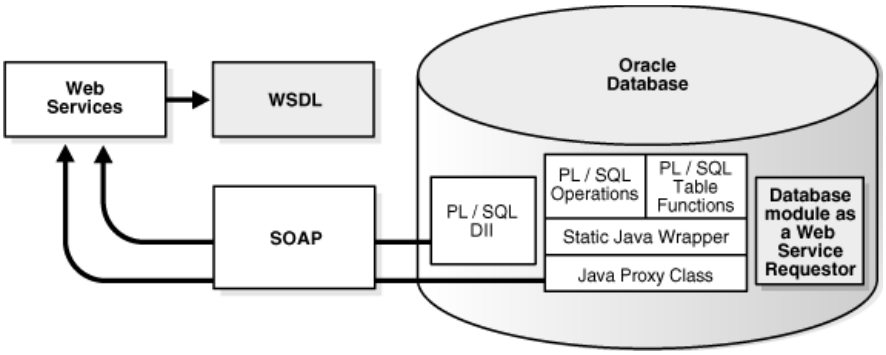
1. A request for a type of database service arrives at the application server. The service endpoint implicitly specifies the type of service requested.
2. The JAX-RPC processing servlet references the SOAP libraries and XML parser to decode the request.
3. The servlet passes the request to the classes that correspond to the exposed database operations. The generated classes can represent PL/SQL packages, queries, DML, AQ Streams, or Java classes in the database.
4. The database passes the response to the JAX-RPC processing servlet, which references the SOAP libraries and XML parser to encode it.
5. A SOAP response formed in accordance with the WSDL is returned to the client.

15.1.2 Database Call-out

You can extend a relational database's storage, indexing, and searching capabilities to include Web services. By calling a Web service, the database can track, aggregate, refresh, and query dynamic data produced on-demand, such as stock prices, currency exchange rates, or weather information. An example of using the database as a service consumer would be to call an external Web service from a predefined database job to obtain inventory information from multiple suppliers, then update your local inventory database. Another example is that of a Web Crawler: a database job can be scheduled to collate product and price information from a number of sources.

The following figure illustrates database call-out.

Figure 15–2 Calling Web Services from Within the Database



The following steps describe the process shown in the previous figure:

- SQL and PL/SQL call specs—Invoke a Web service through a user-defined function call either directly within a SQL statement or view, or through a variable.
- Dynamic Web service invocation using the `UTL_DBWS` PL/SQL package. A Call object can be dynamically created based on a WSDL and subsequently, Web services operations can be invoked.

  Oracle Database PL/SQL Packages and Types Reference provides more information on using the `UTL_DBWS` PL/SQL package.
- Pure Java static proxy class—Generate a client proxy class which uses JAX-RPC. This method simplifies the Web service invocation as the location of the service is already known without needing to look up the service in the UDDI registry. The client proxy class does all of the work to construct the SOAP request, including marshalling and unmarshalling parameters.
- Pure Java using DII (dynamic invocation interface) over JAX-RPC—Dynamic invocation provides the ability to construct the SOAP request and access the service without the client proxy.

Which method to use depends on whether you want to execute from SQL or PL/SQL, from Java classes, or whether the service is known ahead of time (static invocation) or only at runtime (DII).

15.2 Type Mapping Between SQL and XML

The following sections describe the type mappings between SQL and XML for call-ins and call-outs when the Web service is known ahead of time (static invocation).

When the Web service is known at runtime you can use only the Dynamic Invocation Interface (DII) or the `UTL_DBWS` PL/SQL package.

15.2.1 SQL to XML Type Mappings for Web Service Call-Ins

In a database Web service call-in, a SQL operation, such as a PL/SQL stored procedure or a SQL statement, is mapped into one or more Web service operations. The parameters to the SQL operation are mapped from SQL types into XML types.

Note: The reason there may be more than one operation is because OracleAS Web services may be providing additional data representation choices for the SQL values in XML, such as different representations of SQL result sets.

The following table illustrates the SQL-to-XML mappings for Web service call-ins. The first column lists the SQL types. The second column of the table, XML Type (Literal), shows SQL-to-XML type mappings for the default literal value of the use attribute. The third column, XML Type (Encoded), shows the mappings for the encoded value of the use attribute. The literal and encoded values refer to the rules for encoding the body of a SOAP message.

Table 15–1 SQL-to-XML Type Mappings for Web Services Call-ins

SQL Type	XML Type (Literal)	XML Type (Encoded)
INT	int	int
INTEGER	int	int
FLOAT	double	double
NUMBER	decimal	decimal
VARCHAR2	string	string
DATE	dateTime	dateTime
TIMESTAMP	dateTime	dateTime
BLOB	byte[]	byte[]
CLOB	String	String
LONG	String	String
RAW	byte[]	byte[]
Primitive PL/SQL indexby table	Array	Array
PL/SQL Boolean	boolean	boolean
PL/SQL indexby table	complexType	complexType
PL/SQL record	complexType	complexType
REF CURSOR (*name*Beans)	Array	Array
REF CURSOR *name*XML)	any	test_xml
REF CURSOR nameMLRowSet	swaRef	test_xml
SQL object	complexType	complexType
SQL table	complexType	complexType
SYS.XMLTYPE	any	test_xml

Note: If National Language Support (also known as "NLS" or "Globalization Support") characters are used in a SQL SYS.XMLTYPE value, they may not be properly handled.

A query or a PL/SQL function returning REF CURSOR will be mapped into the three methods listed below, where *name* is the name of the query or the PL/SQL function.

- *name*Beans—This method returns an array, where each element is an instance of an XSD complex type that represents one row in the cursor. A complex type sub-element corresponds to a column in that row.

- *name*XMLRowSet—This method returns a swaRef or text_xml response that contains an OracleWebRowSet instance in XML format.
- *name*XML—this method returns an XML any or text_xml response that contains an Oracle XDB row set.

Both OUT and IN OUT PL/SQL parameters are mapped to IN OUT parameters in the WSDL file.

Note that Table 15–1 provides two different mappings: one for literal and another for encoded use. The default mapping is literal. From a database Web service's perspective, there is no special reason why encoded should be used. The mapping for encoded is provided in case you encounter scenarios which call for the encoded use setting. All of the descriptions in this chapter assume that you will be using the literal use setting unless otherwise specified.

15.2.2 XML-to-SQL Type Mapping for Web Service Call-outs

In database Web services call-outs, XML types are mapped into SQL types. The following table lists the XML-to-SQL type mappings used in call-outs.

Table 15–2 XML-to-SQL Type Mappings for Web Service Call-outs

XML Type	SQL Type
int	NUMBER
float	NUMBER
double	NUMBER
decimal	NUMBER
dateTime	DATE
String	VARCHAR2
byte[]	RAW
complexType	SQL OBJECT
Array	SQL TABLE
test_xml	XML Type

A

Pre-Packaged WS-Policy Files for Reliable Messaging

This appendix describes the pre-packaged WS-Policy files that contain typical reliable messaging assertions that you can use to support reliable messaging with WebLogic Java API for XML-based RPC (JAX-RPC) Web services.

This appendix includes the following topics:

- Section A.1, "DefaultReliability1.1.xml (WS-Policy File)"
- Section A.2, "Reliability1.1_SequenceTransportSecurity.xml (WS-Policy File)"
- Section A.3, "Reliability1.1_SequenceSTR.xml (WS-Policy File)"
- Section A.4, "Reliability1.0_1.1.xml (WS-Policy.xml File)"
- Section A.5, "DefaultReliability.xml (WS-Policy File) [Deprecated]"
- Section A.6, "LongRunningReliability.xml (WS-Policy File) [Deprecated]"

You cannot change these pre-packaged files. If their values do not suit your needs, you must create your own WS-Policy file. See Section 8.5, "Creating the Web Service Reliable Messaging WS-Policy File" for details. See "Web Service Reliable Messaging Policy Assertion Reference" in *WebLogic Web Services Reference for Oracle WebLogic Server* for reference information about the reliable messaging policy assertions.

A.1 DefaultReliability1.1.xml (WS-Policy File)

The `DefaultRealiability1.1.xml` WS-Policy file specifies policy assertions related to quality of service. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.pdf`.

```
<?xml version="1.0"?>

<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  >
  <wsrmp:RMAssertion
    xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702"
    >
    <wsrmp:DeliveryAssurance>
      <wsp:Policy>
        <wsrmp:ExactlyOnce />
      </wsp:Policy>
    </wsrmp:DeliveryAssurance>
```

```
  </wsrmp:RMAssertion>
</wsp:Policy>
```

A.2 Reliability1.1_SequenceTransportSecurity.xml (WS-Policy File)

The `Reliability1.1_SequenceTransportSecurity.xml` file specifies policy assertions related to transport-level security and quality of service. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.pdf.

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
    <wsrmp:RMAssertion
         xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
    <wsrmp:SequenceTransportSecurity/>
    <wsrmp:DeliveryAssurance>
      <wsp:Policy>
        <wsrmp:ExactlyOnce/>
      </wsp:Policy>
    </wsrmp:DeliveryAssurance>
  </wsrmp:RMAssertion>
</wsp:Policy>
```

A.3 Reliability1.1_SequenceSTR.xml (WS-Policy File)

The `Reliability1.1_SequenceSTR.xml` file specifies that in order to secure messages in a reliable sequence, the runtime will use the `wsse:SecurityTokenReference` that is referenced in the `CreateSequence` message. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.pdf.

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
  <wsrmp:RMAssertion
        xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
    <wsrmp:SequenceSTR/>
    <wsrmp:DeliveryAssurance>
      <wsp:Policy>
        <wsrmp:ExactlyOnce/>
      </wsp:Policy>
    </wsrmp:DeliveryAssurance>
  </wsrmp:RMAssertion>
</wsp:Policy>
```

A.4 Reliability1.0_1.1.xml (WS-Policy.xml File)

The `Reliability1.0_1.1.xml WS-Policy.xml` file combines 1.1 and 1.0 WS-Reliable Messaging policy assertions. This sample relies on smart policy selection to determine the policy assertion that is applied at runtime. For more information about smart policy selection, see Section 8.5.3, "Using Multiple Policy Alternatives".

> **Note:** The 1.0 Web service reliable messaging assertions are prefixed by `wsrmp10`.

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <wsrmp10:RMAssertion
       xmlns:wsrmp10="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
        <wsrmp10:InactivityTimeout Milliseconds="600000"/>
        <wsrmp10:BaseRetransmissionInterval Milliseconds="3000"/>
        <wsrmp10:ExponentialBackoff/>
        <wsrmp10:AcknowledgementInterval Milliseconds="200"/>
      </wsrmp10:RMAssertion>
    </wsp:All>
    <wsp:All>
      <wsrmp:RMAssertion
           xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
        <wsrmp:SequenceSTR/>
        <wsrmp:DeliveryAssurance>
          <wsp:Policy>
            <wsrmp:ExactlyOnce/>
          </wsp:Policy>
        </wsrmp:DeliveryAssurance>
      </wsrmp:RMAssertion>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

A.5 DefaultReliability.xml (WS-Policy File) [Deprecated]

This WS-Policy file is deprecated. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion Version 1.0 at http://schemas.xmlsoap.org/ws/2005/02/rm/policy/. In the current release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration.

The `DefaultReliability.xml` WS-Policy file specifies typical values for the reliable messaging policy assertions, such as inactivity timeout of 10 minutes, acknowledgement interval of 200 milliseconds, and base retransmission interval of 3 seconds.

```
<?xml version="1.0"?>

<wsp:Policy
   xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
   xmlns:beapolicy="http://www.bea.com/wsrm/policy"
  >

  <wsrm:RMAssertion >

    <wsrm:InactivityTimeout
        Milliseconds="600000" />
    <wsrm:BaseRetransmissionInterval
        Milliseconds="3000" />
    <wsrm:ExponentialBackoff />
    <wsrm:AcknowledgementInterval
        Milliseconds="200" />
    <beapolicy:Expires Expires="P1D" optional="true"/>
  </wsrm:RMAssertion>

</wsp:Policy>
```

A.6 LongRunningReliability.xml (WS-Policy File) [Deprecated]

This WS-Policy file is deprecated. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion Version 1.0 at `http://schemas.xmlsoap.org/ws/2005/02/rm/policy/`. In the current release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration.

The `LongRunningRelibility.xml` WS-Policy files specifies values that are similar to the `DefaultReliability.xml` WS-Policy file, except that it specifies a much longer activity timeout interval (24 hours). See Section A.6, "LongRunningReliability.xml (WS-Policy File) [Deprecated]".

```
<?xml version="1.0"?>

<wsp:Policy
   xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
   xmlns:beapolicy="http://www.bea.com/wsrm/policy"
  >

  <wsrm:RMAssertion >

    <wsrm:InactivityTimeout
        Milliseconds="86400000" />
    <wsrm:BaseRetransmissionInterval
        Milliseconds="3000" />
    <wsrm:ExponentialBackoff />
    <wsrm:AcknowledgementInterval
        Milliseconds="200" />
    <beapolicy:Expires Expires="P1M" optional="true"/>
  </wsrm:RMAssertion>

</wsp:Policy>
```